

ETDK-dolgozat

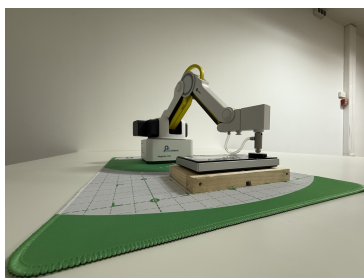
Daroczi Levente
Demeter Beniamin

XXVIII. reál- és humántudományi Erdélyi Tudományos Diákköri Konferencia (ETDK)

Informatika II.: innovatív számítástechnikai termékek, alkalmazások szekció

Kolozsvár, 2025. május 15–18.

Interakciót igénylő eszközök automata tesztelése robotkarral és objektum-felismeréssel



Szerzők:

Daroczi Levente

III. év, Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar

Demeter Beniamin

III. év, Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar

Témavezetők:

dr. Sulyok Csaba, egyetemi adjunktus,

Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar

Katona Gyöngyi-Beáta, szoftvertesztelő, Softech SRL

Mihály Dénes, szoftverfejlesztő, Softech SRL

Szász Arnold-Levente, szoftverfejlesztő, Softech SRL



Interakciót igénylő eszközök automata tesztelése robotkarral és objektum-felismeréssel

Daroczi Levente, III. év, Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar
Demeter Beniamin, III. év, Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar

dr. Sulyok Csaba, egyetemi adjunktus,
Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar
Katona Gyöngyi-Beáta, szoftverteresztelő, Softech SRL
Mihály Dénes, szoftverfejlesztő, Softech SRL
Szász Arnold-Levente, szoftverfejlesztő, Softech SRL

A felhasználói interfészek tesztelése alapvető fontosságú a megbízhatóság és a megfelelő működés biztosítása érdekében. Sok szoftver esetében azonban nem áll rendelkezésre olyan keretrendszer, amely támogatná az automatizált tesztelést, különösen akkor, ha az adott rendszer manuális bevitt igénylő vezérlőkkel, például fizikai gombokkal rendelkezik. Jelenleg ezeknek az eszközöknek a tesztelése nagyrészt manuálisan történik, ami rendkívül időigényes, nagyfokú figyelmet igényel, és növelheti a hibalehetőségeket.

A projekt célja egy olyan automatizált tesztelési rendszer létrehozása, amely robotkart és objektum felismerési technológiát alkalmaz. A rendszer képes fizikai gombokkal és mechanikus vezérlőelemekkel rendelkező eszközök tesztelésére, biztosítva a precíz és ismételhető interakciókat, valamint az eszköz visszajelzéseinek a validálását. Ezzel csökkenthető a manuális beavatkozás szükségessége, miközben jelentősen növelhető a tesztelési folyamat hatékonysága.

A megoldás egy webalkalmazásból és egy backend szerverből áll. A webalkalmazás biztosítja a felhasználói felületet, amely lehetővé teszi a tesztek létrehozását és az eredmények megtekintését, míg a backend szerver felelős a tesztek végrehajtásáért, azaz a robotkar vezérléséért és az objektum-felismerési folyamatok végrehajtásáért.

Tartalomjegyzék

1. Bevezető	1
2. Interfész elemek felismerése képeken	4
2.1. Elméleti háttér	5
2.2. Címkézés és tanítási folyamat	8
2.3. Felismerési folyamat	13
3. Robotkar	15
3.1. Dobot Magician Lite	15
3.2. Kamera	17
3.3. Szoftveres kapcsolat	20
3.4. Kalibráció	23
4. Tesztek	26
4.1. Alkalmazások és hardver eszközök működésének gráfja	26
4.2. Tesztek felépítése	28
4.3. Tesztek futtatása	30
4.4. Tesztek ellenőrzése	32
5. Architektúra	34
5.1. Feldolgozó réteg	34
5.2. Megjelenítési réteg	35
6. Alkalmazás működése	36
7. Következtetések és továbbfejlesztési lehetőségek	40

1. Bevezető

A szoftverfejlesztés világában a kész termék integrált tesztelése legalább olyan fontos, mint az egyes komponensek különálló ellenőrzése. Míg az utóbbi biztosítja az alkotóelemek helyes működését, az előbbi garantálja azok megfelelő együttműködését.[5] A tesztelés ugyanakkor jelentősen bonyolultabbá válik, ha a szoftver nem egy általános célú számítógépen fut, hanem dedikált hardverre készül. [12] Az ilyen esetekben gyakran hiányzik a standard tesztelési infrastruktúra, különösen a mikrokontroller-alapú eszközök esetében. [16] Ilyen támogatás nélkül nehéz, vagy akár lehetetlen, automatikus tesztutasításokat kiadni, illetve azok eredményét visszanyerni a célhardvertől. A kihívást tovább növeli, ha az eszköz fizikai, emberi interakciót igénylő bemeneti vezérlőkkel rendelkezik, mint például nyomógombok, érintőképernyők vagy tekerhető kapcsolók. Ezek tesztelése hagyományos eszközökkel rendkívül nehézkes lehet, különösen a végtermék gyártásának befejezése után. [1] [19]

Egy általános célú tesztelési keretrendszer kialakítása, amely többféle hardvert támogat, rendkívül idő- és erőforrás-igényes lenne, a technológiai változások gyors üteme miatt pedig fenntarthatatlanná válna. A dolgozat ezért egy alternatív megközelítést alkalmaz: a célhardvert feketedobozként kezeli, és a hangsúlyt a fizikai vezérlők és a szoftveres válaszreakció közötti kapcsolat tesztelésére helyezi. Más szóval, egy fizikai bemenet, például egy gomb lenyomása, akkor tekinthető sikeresnek, ha az elvárt szoftveres eseményt kiváltja.

A megvalósítás során három nem szokványos eszköz nyújt támogatást: egy robotkar, egy kamera, valamint egy fejlett objektum-felismerő szoftver. A robotkar az emberi interakció szimulálásáért felelős: fizikai bemeneteket kezel, mint például gombnyomások, érintőképernyős navigáció, vagy tekerőgombok elforgatása. Az objektum-felismerő egység a robotkar irányítását segíti, valamint lehetővé teszi az eredmények ellenőrzését is. Elsődleges feladata a tesztelendő eszköz komponenseinek felismerése a kameraképen keresztül.

A hagyományos tesztelési módszerektől eltérően, a robotkar használata elengedhetetlen az olyan eszközök esetében, amelyek emberi interakciót igénylő fizikai bemeneti interfészekkel rendelkeznek. Bár léteznek különféle tesztelési keretrendszerek, mint például az *Android Debug Bridge (ADB)*¹ Androidos eszközökre, vagy *UI/UX* tesztelésre specializált szoftverek, ezek a megoldások tipikusan csak egy adott platformra vagy bemeneti módszerre korlátozódnak. Hiányzik egy általánosan alkalmazható rendszer, amely képes különböző beviteli interfészekkel (például érintőképernyő, fizikai gomb, tekerőkapcsoló) rendelkező eszközöket kezelni.

¹Forrás: <https://developer.android.com/tools/adb>, utolsó megtekintés dátuma: 2025-04-22

A teszteszköz komponenseinek megfelelő leírásával, valamint a mély tanulással betanított felismerő modellek segítségével teljes, több lépésből álló tesztek hajthatók végre emberi beavatkozás nélkül. Az automatizált rendszer bevezetése számos előnnyel jár a hagyományos, manuális teszteléssel szemben. Egyrészt kiküszöbölhetőek az emberi hibák, másrészt jelentős idő szabadítható fel a tesztmérnökök számára.[1] A rendszer képes önállóan elvégezni a tesztelést, így a felhasználónak elegendő csupán a futtatás végén áttekinteni az eredményeket és a kamerafelvételt, amely dokumentálja a tesztkörnyezet állapotát.

A rendszer nem teljesen automatizált, ami elsőre hátránynak tűnhet, de valójában előnyt jelenthet bizonyos helyzetekben. Fontos, hogy az eredmények felügyelet mellett szülessenek, és egy esetleges hiba esetén a rendszer képes legyen azt megfelelően kezelni. Így elkerülhető, hogy ugyanaz a hibás teszt feleslegesen fusson újra és újra. Ez az irány kontrolláltabb és biztonságosabb működést tesz lehetővé, miközben megmarad a produktív tesztelés és az időhatékony működés közti egyensúly.

A piacon jelenleg kevés olyan megoldás létezik, amely robotkar és objektum-felismerő technológia kombinálásával képes interakciót igénylő eszközök szoftveres tesztelésére. A legismertebb hasonló rendszerek közé tartozik a *MATT*², melyet az *Adapta Robotics* fejlesztett, valamint a *QUACO Pro*³ a *Sastra Robotics* portfóliójából. Ezek a megoldások saját fejlesztésű robotkarokat használnak, amelyek speciálisan erre a célra lettek kialakítva. Ebből adódóan bizonyos esetekben hatékonyabbak lehetnek, mint az általános célú robotkarra épülő rendszerek.

Fejlesztés és munkamenet

A projekt első fázisa a *Codespring* szoftverfejlesztő cég nyári gyakorlata során indult el, egészen pontosan a 2024-es év nyarán. A fejlesztés körülbelül három hónapon át zajlott, a két szerző közös munkájának eredményeként. A cég célja az volt, hogy a gyakornokok valós projekt tapasztalatot szerezzenek, ennek érdekében három mentort rendelt a projekthez: Katona Gyöngyi, Mihály Dénes és Szász Arnold személyében. Ők szakmai tudásukkal, rendszeres kódellenőrzéssel, valamint a szükséges erőforrások biztosításával járultak hozzá a fejlesztés sikeréhez. A projekt koordinációját Dr. Sulyok Csaba és Kelemen-Fehér Bálint végezte, akik szintén értékes tanácsokkal és megoldási javaslatokkal támogatták a munkát. Az említett *Codespring* munkatársak nemcsak a gyakorlat ideje alatt, hanem az azt követő fejlesztési

²Forrás: <https://www.adaptarobotics.com/matt/>, utolsó megtekintés dátuma: 2025-04-22

³Forrás: <https://sastrarobotics.com/products/quaco-pro/>, utolsó megtekintés dátuma: 2025-04-22

szakaszokban is aktívan részt vettek a projektben, ezért ezúton is hálás köszönet illeti őket. A nyári gyakorlat keretében sikerült meghatározni a projekt célját, kialakítani a meghatározó funkcionalitásokat, valamint bemutatni az elkészült prototípust a gyakorlat zárásakor. Ebbe beletartozott az első teszteszköz felvezetése, a robotkar okostelefonra szabott vezérlése, a tesztek végrehajtásának logikája, valamint más kulcsfontosságú rendszerfunkciók megvalósítása.

Megragadva az alkalmat, köszönet illeti Lokodi Máriát, a *Codespring* munkatársát, aki az alkalmazás felhasználói felületének dizájnját tervezte meg.

A projekt következő fázisa a *Csoportos Projekt* tantárgy keretein belül valósult meg. Megközelítőleg négy hónap alatt, hat fejlesztő közös munkájának eredményeként számos új, jelentős funkcionalitással bővült az alkalmazás. Többek között egy új teszteszköz felvezetése, a felhasználómenedzsment integrálása, valamint a robotkar kalibrációs adatainak importálási és exportálási lehetősége gazdagította a meglévő rendszert. A projekt ezen kiegészítésekkel együtt sikeresen bemutatásra került a félév végi értékelés során.

Az említett új funkcionalitások nem valósulhattak volna meg Boda Norbert, Dáczy Krisztián, Kósa Mátyás, illetve Póti Ábel segítségével nélkül, akik mindannyian csatlakoztak a *Csoportos Projekt* alatti fejlesztéshez, köszönet illeti őket munkájukért.

2. Interfész elemek felismerése képeken

Mivel bármilyen felhasználói interfészen való navigálás a cél, ezért fontos, egy olyan technológia választása amely könnyedén lehetővé teszi ezt. A felhasználói interfészekben a navigálás különböző gombok, érintőképernyő segítségével történik, a kihívás az volt, hogy olyan technológia legyen használva amely képes ezeket az interaktív felületeket, felismerni.

Az elsőként bevezetett felhasználói felület egy mobilalkalmazás volt, a *FestivApp*⁴, amelyet a *Codespring* fejleszt. Ez az alkalmazás lehetőséget biztosít a felhasználók számára, hogy böngésszenek különböző fesztiválok programjai között, megkönnyítve az események áttekintését.

Az alkalmazáson belül azok az objektumok, amelyekkel a felhasználók kapcsolatba léphetnek, specifikus formával rendelkeznek, a legtöbb esetben négyzet alakúak. Ezek az interaktív elemek segítik az intuitív navigációt és a könnyebb használhatóságot, mivel egyértelműen elkülönülnek a statikus tartalmaktól.

Ezért az első lehetőséget a számítógépes látás jelentette. Az *OpenCV*⁵ egy olyan szoftver amely lehetőséget nyújt ahhoz, hogy különböző algoritmusok fussanak le, amelyek segítenek valamilyen specifikus alakú objektumokat azonosítani. Ez a megoldás viszont nem kedvező más felhasználói felület esetében, mivel nem minden esetben négyzet alakúak, vagy más mértani alakzatúak a gombok. Van olyan applikáció amely esetében egy ikon és a hozzá tartozó címke jelent egy gombot és ezek nincsenek semmiféle mértani alakzattal körbevéve [8].

A második opció az OCR (optikai karakterfelismerés). A mobil alkalmazás minden gombja rendelkezett egy hozzá tartozó címkével, ezért felmerült annak a lehetőségnek az alkalmazása, hogy karakterfelismerő algoritmusok által ismerjük fel ezeket. Ez az opció jobb mint az előző, de ebben az esetben is fennáll annak a lehetősége, hogy nem minden esetben vannak címkézve a gombok [4].

Az utolsó opciót az objektum-felismerő algoritmusok jelentik, amelyek képesek különböző formákat, ikonokat és akár rövidebb szöveget is felismerni. Az objektum-felismerő algoritmusok képesek az egyes vizuális minták és struktúrák detektálására, amelyek segítségével a gombokat, felületeket és más elemeket lehet azonosítani anélkül, hogy a rendszernek előre meghatározott geometriai formákkal kellene rendelkeznie.[10]

⁴Forrás: <https://festivapp.eu/>, utolsó megtekintés dátuma: 2025-04-19

⁵Forrás: <https://opencv.org/>, utolsó megtekintés dátuma: 2025-04-22

2.1. Elméleti háttér

Az objektum-felismerés mélytanuláson alapszik, és lehetővé teszi különböző objektumok, tárgyak azonosítását és helymeghatározását képeken vagy videókon. A célja az objektum-felismerő algoritmusoknak az, hogy egy objektumot detektáljon, ennek a helyét meghatározza és egy osztálycímkével lássa el.[14]

Gépi tanuláson alapuló objektum-felismerő modellek olyan modellek, amelyek képesek felismerni objektumokat. Ezek a modellek különböző jellemzőket nyernek ki, több feldolgozó réteg segítségével. Ahhoz, hogy ezek a modellek képesek legyenek felismerni objektumokat/tárgyakat, címkézett adatokra van szükség, amelyek minden felismerni kívánt tárgyhoz rendelnek egy osztályt/címkét. A modellek ezeket a felcímkézett adatokat használják annak érdekében, hogy minden osztálynak megfelelő jellemzőt megtanuljon. Gépi tanuláson alapuló objektum-felismerő modellek például az SVM (support vector machines) vagy a döntési fák (decision tree).

A mélytanuláson alapuló modellek olyan neurális hálózatok osztályát jelentik, amelyek automatikusan képesek felismerni objektumokat, a gépi tanuláson alapuló modellekhez hasonlóan. A különböző rétegek amelyek jellemzőket nyernek ki tartalmazhatnak konvolúciós neurális hálózatot (CNN), régió alapú konvolúciós neurális hálózatot (R-CNN), úgy nevezett egylövéses detektorokat (single-shot detector SSD), vagy YOLO (You Only Look Once) modelleket, amelyek képesek több objektum felismerésére is. Annak érdekében, hogy mélytanuláson alapuló modellek jól teljesítsenek nagy adathalmazra van szükség, amely olyan képeket tartalmaz amelyeken minden felismerni kívánt objektum egy kerettel és egy osztállyal/címkével van ellátva. A modell megtanulja az objektumok felismerését, azáltal, hogy minimalizálja a veszteség függvényt (loss-function), amely az előre jelzett, és valódi címkék és keretek közötti eltérést méri. [11]

A mélytanuláson alapuló objektum-felismerő modellek esetében az objektumok kategóriáinak azonosítása kulcsfontosságú lépés. Az objektum-felismerés különböző megközelítéseket kínál a tárgyak pontos azonosítására és lokalizálására képeken vagy videókon: Two-stage detektorok, Single-stage detektorok, Anchor Free detektorok. A Two-stage detektorok jelentették az első megközelítéseket amelyeket az objektum-felismerést tették lehetővé, viszont napjainkban ritkán használják. Ezek a módszerek először lehetséges tárgyakat azonosítanak úgynevezett *region proposal* technikákkal. Az így azonosított régiókat egységes méretre átméretezik, és a javaslatokat külön batch-elemként kezelik. Az ismert algoritmusok közé

tartoznak például a Fast R-CNN, a Faster R-CNN és a Mask R-CNN. Ezzel szemben az egylépcsős (Single-stage) detektorok minden pozícióhoz előre rögzített javasolt kereteket, úgynevezett *anchor box*-okat rendelnek. A modell kimenete ezeket az előre meghatározott kereteket pontosítja és javítja ki. Ezek a módszerek gyorsak és kellően pontosak, ezért széles körben elterjedtek. Példák erre a YOLO (You Only Look Once), SSD (Single Shot Multibox Detector). Az Anchor Free detektorok a legújabb megközelítést képviselik, amelyek még nem terjedtek el széles körben, de már most kiemelkedő eredményeket érnek el. A predikció egy valószínűségi hőtérkép amely jelzi, hogy milyen pozíción a legvalószínűbb egy tárgy középpontja (például a CenterNet). [17]

A különböző objektum-felismerő technológiák közül a legmegfelelőbb választás a YOLO (You Only Look Once) algoritmuscsalád, mivel ez a megközelítés képes egyetlen képen több objektumot is azonosítani. Ez különösen fontos egy felhasználói interfészen, ahol a valós idejű és pontos detektálás kulcsszerepet játszik. A YOLO egy Single-stage detektor, amely az évek során számos fejlesztésen ment keresztül, folyamatosan integrálva a legújabb módszereket az architektúrájába. A projekt során a választás a YOLOv8 verzióra esett, mivel ez volt a legújabb és legfejlettebb változat a fejlesztés idején. Ezen felül egy további fontos előny az *Ultralytics* Python csomag használata, amely lehetőséget biztosít a YOLO modellek betanítására és alkalmazására Python környezetben, ez különösen előnyös, mivel a teljes feldolgozóréteg is Python nyelvben íródott.

A YOLO az objektumokat és azok pozícióját "egyetlen pillantással", azaz egyszeri képfeldolgozással ismeri fel, ezért kapta a *You Only Look Once* nevet. A YOLO háttérben egy CNN-t (konvolúciós neurális hálózatot) használ, ami segít abban, hogy egy képen különböző objektumok keretét és a hozzájuk tartozó valószínűségét előrejelezze. A konvolúciós neurális hálók nagyon hatékonyak vizuális adatok feldolgozásában, ugyanis a kezdeti konvolúciós rétegekből a jellemzők hatékonyan tudnak átjutni későbbi rétegekbe [9].

A felismerés során a kép egy $N \times N$ méretű rácshálóra lesz felosztva, ahol minden egyes rácscella felelős azért, hogy felismerje, ha egy objektum középpontja az adott cellán belül található. Minden cella B darab keretet (*bounding box*-ot) jósol meg, beleértve azok pozícióját és méreteit. Ezen kívül a modell kiszámítja az objektum jelenlétének valószínűségét és az osztályhoz tartozó valószínűségeket (lásd cella attribútumai 1. táblázat).

Egy cella $(B \times 5 + C)$ értéket jósol, így a teljes kimeneti tenzor mérete $N \times N \times (B \times 5 + C)$, ahol C az osztályok száma. Az egyes keret bizalmi érték (confidence score) az p_c és

Cella attribútumai	
p_c	Annak valószínűsége, hogy a cellában egy objektum található.
(b_x, b_y)	A keret középpontjának koordinátái.
(b_w, b_h)	A keret szélessége és magassága.
$p(c_i)$	Annak valószínűsége, hogy az objektum az i -edik osztályba tartozik, feltéve, hogy az objektum jelen van.

1. táblázat. Cella attribútumai.

az *Intersection over Union (IoU)* szorzataként adódik, amely az előre jelzett és a valós keret átfedését méri. Ha egy cellában nincs objektum, a bizalmi pontszám nulla lesz. Minden kerethez tartozik egy osztályspecifikus érték, c_{ss} (class specific score), amely kódolja az adott osztály valószínűségét a kereten belül, valamint azt, hogy mennyire illeszkedik a keret az objektumhoz. [6]

Mivel az objektumok, amelyeket fel kell ismerni, különböző méretűek, az algoritmus számára szükséges keretek, azaz az ún. *anchor box*-ok, szintén különböző méretűek és arányúak. Ezek az *anchor box*-ok előre meghatározott minták alapján kerülnek meghatározásra, hogy lefedjék az objektumok különböző méreteit és arányait a tanulóhalmaz alapján. Minden ilyen kerethez tartozik két paraméter: egy magasság (p_h) és egy szélesség (p_w). Az *anchor box*-ok, a tanulási adathalmazban található keretek klaszterezéséből vannak meghatározva, így ezek a keretek az adatok eloszlásához igazodnak. Annak érdekében, hogy a modell képes legyen felismerni az objektumokat különböző skálákon, minden egyes hálózati réteghez több különböző méretű és arányú *anchor box* kerül társításra. Ez lehetővé teszi, hogy a modell alkalmazkodjon az objektum méretéhez anélkül, hogy teljesen új kereteket kellene létrehozni minden egyes objektumhoz.

Ugyanakkor szomszédos rácscellák ugyanazt az objektumot jósolhatják, ami azt jelenti, hogy egy objektumhoz több átfedő keret is tartozhat. Emiatt előfordulhat, hogy egy objektumhoz több predikció is tartozik. Annak érdekében, hogy ezeket a kereteket minimalizálni lehessen két szűrési feltétel van alkalmazva. Az első szűrési feltétel abban áll, hogy minden rácshoz tartozó kerethez egy osztályspecifikus pontszám, a keret koordinátái és egy kategóriába sorolási kimenet kapcsolódik. Összesen $N^2 \times B$ előre jelzett keret lesz, és azokat a kereteket, amelyek osztályspecifikus pontszáma egy előre meghatározott küszöbérték alatt van, el kell vetni (általában 0.5, de ez adathalmaztól függ). A második szűrési feltétel az úgy nevezett *non-max suppression - NMS*. Az algoritmus az *Intersection over Union (IoU)* mértéket használja, amely két határolókeret metszetének és egyesítésének arányát fejezi ki. Az *NMS* először sorba rendezi a detektált kereteket a konfidencia értékük (confidence score) alapján, majd iteratíván eltávolítja



1. ábra. Non-Max Suppression vizualizáció

azokat a kereteket, amelyeknek az IoU értéke egy küszöbérték fölött van, így elkerülve a duplikációt (lásd 1. ábra ⁶). [6]

$$IoU = \frac{Keret_1 \cap Keret_2}{Keret_1 \cup Keret_2}$$

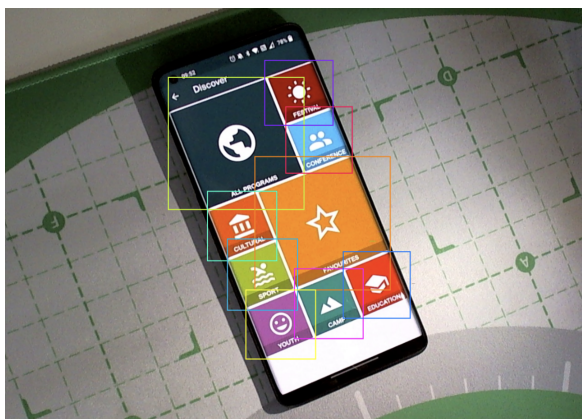
2.2. Címkézés és tanítási folyamat

A modellbetanítás és objektum-felismerés során az első lépés egy jól strukturált és megfelelően címkézett adathalmaz létrehozása. A felhasználói interfészek (UI) elemeinek felismerése során az volt a tapasztalat, hogy a különböző alkalmazások interfész-elemei között jelentős eltérések lehetnek. Például egy mobilalkalmazásban sokféle gomb, ikon és interaktív elem található, amelyek vizuálisan és funkcionálisan is eltérhetnek egymástól.

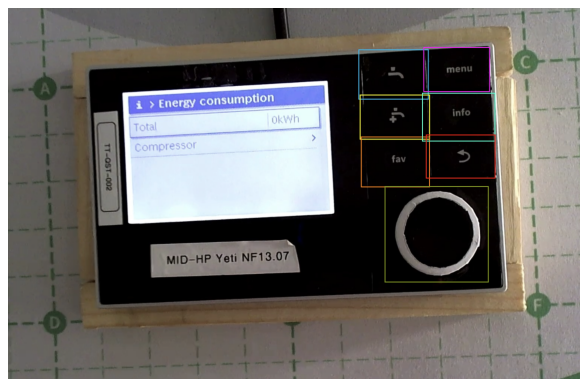
A mobil alkalmazás (*FestivApp*) interakciós elemei esetén észre lehetett venni némi hasonlóságot, ezért az a döntés született, hogy ezen hasonlóságok szerint minden gomb csoportosításra kerüljön. A fő menüoldalról a gombokat, navigálásra szolgáló gombok, oldalfülek és különböző akciót végrehajtó gombok, például kedvencekhez hozzáadás, külön-külön modellekre legyenek csoportosítva.

Ezzel szemben a fűtőrendszerek vezérlésére használt kontroller esetében a felhasználói felület sokkal egyszerűbb: kevés interakciós elem szerepel rajta, és ezek vizuálisan is nagyon hasonlítanak egymásra. Ennek következtében elegendő egyetlen modell betanítása, amely az

⁶Forrás: <https://thepythoncode.com/article/non-maximum-suppression-using-opencv-in-python>, utolsó megtekintés dátuma: 2025-04-20



(a) FestivApp fő menüoldala címkékkel ellátva.



(b) Fűtőrendszerek vezérlésére használt controller gombjai címkékkel ellátva.

2. ábra. Két különböző teszteszköz címkézett komponensei.

összes gombot és vezérlőelemet felismeri.

Fontos kiemelni, hogy a modellek létrehozása és integrálása a fejlesztők feladata, mivel a jelenlegi rendszerben a felhasználóknak még nincs lehetőségük új eszközök hozzáadására. Az előre betanított modellek tehát fixen meghatározott interfészekre lettek optimalizálva.

Annak érdekében, hogy a modellek minden körülmény között pontosan és megbízhatóan felismerjék a kívánt objektumokat, az adathalmazt változatos környezetben kellett létrehozni. Ez azt jelentette, hogy a képeket különböző fényviszonyok mellett kellett rögzíteni, folyamatosan változtatva a fényforrásokat, valamint a tesztelendő eszközt több szögből és különböző pozíciókban kellett lefotózni. Emellett a kamera elhelyezkedése is rendszeresen módosítva volt, hogy a modell képes legyen kezelni a perspektívabeli eltéréseket.

A rögzített képek pontos annotálása (címkézése) elengedhetetlen a hatékony tanításhoz. Ehhez a *Roboflow*⁷ platform kerül használatra, amely lehetőséget biztosít adathalmazok létrehozására és kezelésére. Az egyik előnye, hogy ezen a felületen a címkézési folyamat csapatban is hatékonyan végezhető, így a munkát párhuzamosan több fejlesztő is el tudja végezni (címkézett képeket lásd 2a. és 2b. ábra). [3]

Az adathalmaz megfelelő előkészítése kulcsfontosságú a modellek hatékony tanításához. Mielőtt az adathalmaz ténylegesen felhasználásra került volna a modell betanításához, három fő fázison ment keresztül: adathalmaz felosztás, előfeldolgozás és augmentáció.

Az adathalmaz felosztása során a képeket három nagy csoportba kell felosztani: tanítási, validációs és teszt adathalmazok. A tanítási adathalmaz az adatok legnagyobb részét jelentik (általában a teljes adathalmaz 70-80%-a), ezen keresztül tanulja meg a modell az objektumok

⁷Forrás: <https://roboflow.com/>, utolsó megtekintés dátuma: 2025-04-25

felismerését és a különböző jellemzők/mintázatok azonosítását. A validációs adathalmaz segíti a modell teljesítményének a felügyelését. Ennek az adathalmaznak a segítségével állíthatók be hiperparaméterek és elkerülhető a túlilleszkedés (overfitting), amely akkor fordul elő, ha a modell túlságosan a tanító adatokhoz igazodik, és kevésbé képes általánosítani. A teszt adathalmaz a végső modell teljesítményét képes értékelni, ez biztosítja, hogy a modell képes új, ismeretlen adatokon is jól teljesíteni.

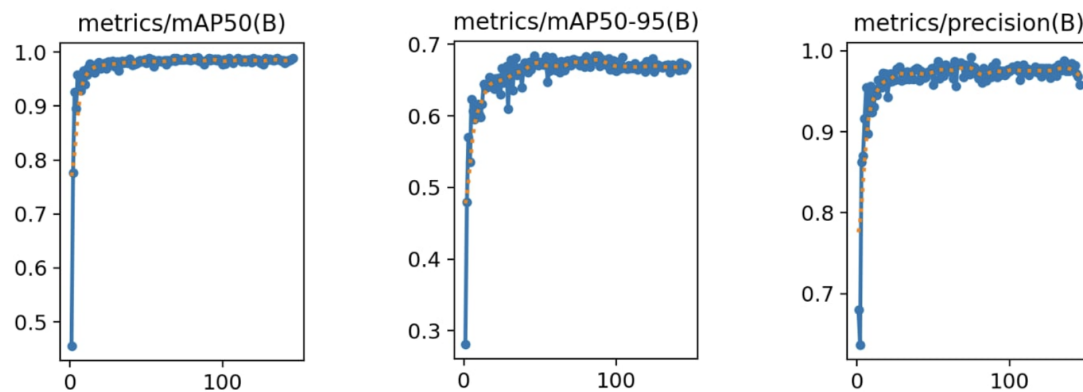
Az előfeldolgozási fázisban a képeken auto-orientáció van alkalmazva, ami biztosítja, hogy a képek mindig azonos tájolással kerüljenek be a tanítási folyamatba. Ugyanakkor átméretezés is van alkalmazva, ez elősegíti a YOLO által használt algoritmust miszerint az eredeti képek $N \times N$ rácscellára lesznek felosztva.

Az adathalmaz gazdagítása érdekében alkalmazva vannak adataugmentációs technikák, amelyek mesterségesen növelik az adathalmaz sokféleségét. Ez segít a modell általánosítási képességének javításában, csökkenti a túlilleszkedés esélyét, és biztosítja, hogy a modell különböző környezetekben és körülmények között is megfelelően működjön. Az augmentációs technikák közé tartozik a forgatás (rotation), amely során a képek különböző szögekben (például $\pm 10-20^\circ$) elforgatásra kerülnek, hogy a modell képes legyen az objektumokat eltérő perspektívákból is felismerni. A fényerő és kontraszt módosítása (brightness/contrast adjustment) lehetővé teszi, hogy a képek fényereje és kontrasztja véletlenszerűen változzon, ami segíti a modellt abban, hogy különböző fényviszonyok között is hatékonyan működjön. Ezen kívül alkalmazható a zaj hozzáadása (noise injection) is, amely hozzájárul a modell robusztusabbá tételéhez, így az képes lesz kezelni a valós adatokban előforduló vizuális hibákat és torzításokat.

A megfelelő augmentációs paraméterek kiválasztása azonban sok kísérletezést igényel. Ha az augmentáció túlzott mértékű, az negatívan befolyásolhatja a modellt, míg a megfelelően beállított augmentáció jelentősen javíthatja a modell felismerési képességét, biztosítva, hogy a modell pontos és megbízható legyen.

A *mAP* (*mean Average Precision*), az objektum-felismerési modellek teljesítményének mérésére szolgáló mutató. Ez az érték egyszerre veszi figyelembe a helyes találatokat (true positives) és a téves találatokat (false positives), így jól tükrözi, hogy a modell mennyire képes pontosan felismerni az objektumokat. [18]

A 3. ábrán látható *mAP50* grafikon azt mutatja, hogy a modell milyen arányban ismeri fel az objektumokat egy 50%-os IoU (Intersection over Union) küszöb mellett. A *mAP50-95* grafikon ennél részletesebb képet ad, mivel több, 50% és 95% közötti átfedési küszöb alapján



3. ábra. A modell teljesítményének alakulása. A grafikonokat a *YOLOv8* tanítási folyamata generálta.

értékeli a modell teljesítményét, így jobban tükrözi annak általános felismerési képességét, és finomhangolási lehetőségeket is feltár. A *Precision* grafikon pedig azt mutatja meg, hogy a modell által pozitívnak jelölt felismerések milyen arányban voltak valójában helyesek, ezzel a felismerések megbízhatóságára világít rá.

Az *epoch*-ok a tanulási folyamatban fontos szerepet játszanak, ugyanis a modell ahányszor végigfut az egész tanítási adathalmazon, frissíti a súlyokat a hibák minimalizálása érdekében. Minden *epoch* során a veszteségi értékek általában csökkennek, míg a pontossági és *mAP* mutatók emelkednek, ami a fejlődést tükrözi. Ezeket a változásokat a grafikonokon is nyomon lehet követni, így látható, hogyan javul a modell a tanulási folyamat során. Ugyanakkor figyelni kell az esetleges túlillesztés jeleire is, amikor a modell elkezd a tanítási adatokra túlságosan illeszkedni, így az *epoch*-ok optimális számának meghatározása fontos szerepet játszik a tanítási folyamatban.

Implementáció szempontjából, az *Ultralytics* csomagot használva, tanítani is lehet a modelleket. A tanítási folyamathoz egy előre definiált csomagstruktúrára van szükség, amelyben a képek és a címkézésben definiált keretek koordinátái kell szerepeljenek szöveges állományokban, minden képhez megfelelően.

A *dataset.yaml* (lásd 5. kódrészlet) fájl tartalmazza a modell számára szükséges alapinformációkat, beleértve a tanítási, validálási és opcionálisan a teszt adathalmaz elérési útját, valamint az osztályok számát és megnevezéseit.

A *test* mappa és az ehhez tartozó útvonaltól opcionális. Ha nincs megadva, a modell nem használja a tesztelési adathalmazt. A *names* kulcsnál a megadott lista az osztályok elnevezését tartalmazza, amelyek a címkézési fájlokban indexelve szerepelnek.



4. ábra. A dataset könyvtárstruktúrája tanítási, test és validációs adatokkal.

```

path: /path/to/dataset # A teljes
    ↳ adathalmaz gyökérkönyvtára
train: train/images # Tanítási
    ↳ képek mappája
val: valid/images # Validációs
    ↳ képek mappája
test: test/images # (Opcionális)
    ↳ Tesztelési képek mappája
nc: 3 # Az osztályok száma (példa:
    ↳ 3 osztály)
names: ["favorites", "camp",
    ↳ "sport"] # Az osztályok nevei
  
```

5. ábra. *dataset.yaml* fájl tartalom példa.

A YOLOv8 támogatja a különböző formátumú adathalmazokat is (például COCO, YOLO, VOC), de a fenti struktúra (4. ábra) a YOLO saját formátuma [20]. A címkézési fájlok (.txt formátumban) a *labels/* mappákban találhatók, és minden képhez tartozik egy azonos nevű .txt fájl, amelyben az alábbi formátumú adatok szerepelnek:

```

<osztály_id> <keret_x_középpont> <keret_y_középpont> <keret_szélessége>
↳ <keret_magassága>
  
```

A *dataset.yaml* helyes beállítása kulcsfontosságú, mivel az *Ultralytics* könyvtár automatikusan ebből olvassa ki a szükséges konfigurációs adatokat, így biztosítva a megfelelő tanítási és validációs folyamathoz szükséges információkat.

A tanítási folyamat (lásd 1. kódrészlet) után az eredményeket tartalmazó mappában különböző statisztikákat bemutató képek és állományok találhatók, amelyek segítenek a modell teljesítményének kiértékelésében. Az egyik legfontosabb fájl a *best.pt*, amely a legjobb teljesítményű modellt tartalmazza. A YOLOv8 tanítási folyamatában az *Ultralytics* könyvtár automatikusan figyeli a modell teljesítményét, és a *best.pt* fájlba mindig azt az állapotot menti

```

from ultralytics import YOLO

model = YOLO("yolov8n.pt")
model.train(
    data="path/to/dataset.yaml",
    epochs=150,
    project=path/to/directory/where/result/will/be/saved,
    name="name/of/result/directory",
)
  
```

1. kódrészlet. Python szkript ami segítségével a tanítási folyamatot lehet futtatni.

```

model = YOLO("best.pt")
results = model(frame)

for result in results:
    if hasattr(result.bboxes, "xyxy"):
        boxes = result.bboxes.xyxy
    else:
        boxes = result.xyxy

    class_ids = result.bboxes.cls
    for box, class_id in zip(boxes, class_ids):
        x1, y1, x2, y2 = box[:4]
        x1, y1, x2, y2 = map(float, (x1, y1, x2, y2))
        class_name = class_names[int(class_id)]
        print(f"Object {class_name} found in the image. Coordinates: {x1},
        ↪ {y1}, {x2}, {y2}")

```

2. kódrészlet. Python szkript ami segítségével egy modellt fel lehet használni, hogy a számítógép kamerájával felvett képen a modell által felismert objektumokat ismerjen fel.

el, amely a legjobb validációs eredményt érte el egy adott *epoch* során. Ezen kívül egy *last.pt* fájl is létrejön, amely az utolsó *epoch* állapotát tárolja, függetlenül attól, hogy az a legjobb volt-e.

2.3. Felismerési folyamat

Mivel a *best.pt* állomány tartalmazza a modellt, ezért ennek segítségével kell felismerni a kívánt objektumokat/elemeket egy képen belül (lásd 2. kódrészlet).

A projekt egy dedikált felismerő réteget tartalmaz, amely felelős különböző objektumok azonosításáért az adott paraméterek alapján. Annak érdekében, hogy ez a réteg ne legyen specifikus egy adott teszteszközhöz, úgy lett kialakítva, hogy további eszközök bevezetése esetén ne legyen szükség módosításokra ezen a rétegen.

Mivel az interfészen belüli interakciós elemek csoportosítva vannak, és minden csoporthoz külön modell tartozik, ezért egy olyan megoldásra van szükség, amely rugalmasan és hatékonyan kezeli az objektum-felismerési folyamatot. A probléma megoldását konfigurációs állományok jelentik, amelyekben az interfész objektumairól és a modellekhez tartozó fájlok elérési útvonaláról tárol információkat.

Minden teszteszközhöz tartozik egy külön mappa, amely az *apps* nevű főmappában található. Ezek a mappák tartalmazzák az adott teszteszközhöz kapcsolódó modelleket és konfigurációs állományokat, biztosítva ezzel a rendszer modularitását és egyszerű bővíthetőségét.

Minden teszteszközhöz tartozik egy *app.json* konfigurációs fájl, amely *JSON* objektumok

formájában tárolja az interfész struktúráját. Az állomány alapértelmezés szerint két fő objektumot tartalmaz az alábbi formátumban:

```
[
  {"id": "elements", "contains": []},
  {"id": "pages", "contains": []}
]
```

Ahol az *elements* objektum tartalmazza az egyes modellek- és a modellekhez tartozó felismerendő objektumok azonosítóját és a *pages* objektum pedig az interfész egyes oldalainak leírását tartalmazza.

Minden modell egy *JSON* objektumként szerepel az *elements* mezőben, ahol a *contains* lista az adott modell által felismerendő objektumok azonosítóit tartalmazza. Példa egy modell definíciójára:

```
{
  "id": "navigation",
  "contains": [
    {
      "id": "back"
      "contains": []
    },
    ...
  ]
}
```

Mivel minden *id* mező egyedi, és az interakciós elemek nevét tartalmazza, a rendszer könnyedén eldöntheti, hogy egy adott objektum melyik modellhez tartozik. Minden objektumcsoporthoz egy külön modell rendelhető, ezáltal biztosítva a felismerő réteg modularitását és bővíthetőségét.

Egy másik konfigurációs állomány a *best_trained_models.json*, amely a modellekhez és a hozzájuk tartozó adathalmazokhoz, konfigurációs állományokhoz és tanítási eredményekhez vezető elérési útvonalakat tartalmazza. Ez lehetővé teszi, hogy a rendszer betöltse a megfelelő modelleket az adott felismerési feladathoz.

A felismerési folyamat során segédfüggvények biztosítják, hogy a konfigurációs állományokban tárolt információk felhasználásával a rendszer rugalmas módon tudjon objektumokat azonosítani. Ez lehetővé teszi, hogy a felismerési réteg bármely teszteszköz esetén könnyedén alkalmazható legyen anélkül, hogy eszközszer specifikus módosításokra lenne szükség.

3. Robotkar

3.1. Dobot Magician Lite

Az alkalmazás a *Dobot Robotics* által fejlesztett, *Magician Lite* nevű, oktatásra szánt robotkart veszi igénybe a teszteszköz vezérlésére. Ez egy általános célú robotkar, melyet hardveresen, szoftveresen illetve Python programok segítségével is lehet irányítani. A kommunikáció USB kábelon keresztül történik, míg az áramellátás a hálózat segítségével valósul meg. [15]

A robotkar 0.2 mm-es pontossággal képes végrehajtani egy utasítást[15], ami elegendő precizitást biztosít még olyan esetekben is, amikor a célpont mérete kifejezetten kicsi. A projekt során minden vezérlőelem és komponens jóval nagyobb méretű ennél, így a robotkar pontossága bőven megfelelőnek bizonyul. A kar végére többféle fej, azaz végrehajtó eszköz csatlakoztatható: például írószer, összehárható markoló egység, vagy egy forgatható tapadókorong, amely vákuumos rögzítésre képes kisebb tárgyak megfogásához (lásd 6. ábra és 2. táblázat).

A robotkar felépítése az emberi karhoz hasonló, szerkezeti elemei különböző mozgási szabadságfokokkal rendelkeznek. A kar egy bázishoz van rögzítve, amelyből kiindul az alkar,



6. ábra. A DOBOT Magician Lite robotkar mindhárom végrehajtó eszközével [15]

Végrehajtó eszközök	
Írószer tartó	Toll átmérő: 8-12 mm
Tapadókorong	Beépített légszivattyús meghajtó, működik negatív nyomás alatt, 10 mm, 20 mm átmérőjű szívócsészével
Puha markoló	Beépített légszivattyús meghajtás, pozitív és negatív nyomás alatt is működik, maximális nyitási és zárási távolsága 50 mm

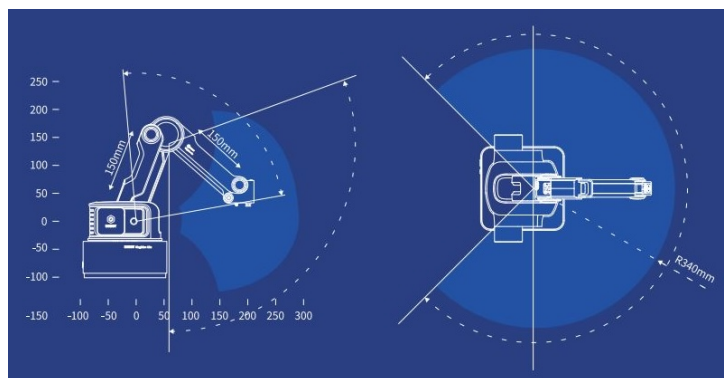
2. táblázat. DOBOT Magician Lite végrehajtó eszközök [15]

amelyet a felkar követ. Az első tengely a bázison helyezkedik el, és lehetővé teszi a robotkar horizontális irányú elforgatását. Az alkar és a felkar függőleges mozgást végezhet, biztosítva ezzel a szükséges rugalmasságot a különböző pozíciók eléréséhez (lásd 7. ábra). Emellett bizonyos végrehajtó eszközök, például a tapadókorongos egység, saját tengelyük körül is el tudnak fordulni, ami tovább növeli a rendszer alkalmazkodó képességét.

A *Dobot Robotics* robotkarját számos fejlett és hasznos funkcióval látták el, különösen a kísérő alkalmazás felhasználói felületén keresztül. A robotkar képes precízen megrajzolni összetett grafikákat, akár 250 gramm tömegű tárgyakat mozgatni, valamint egyéb hasonlóan összetett műveleteket végrehajtani. A dolgozatban azonban egy specifikus célra kerül felhasználásra, amelyre a gyártó nem biztosít közvetlen megoldást. Ennek eredményeként alacsony szintű műveletekből kellett kiindulni, például a robotkar adott háromdimenziós koordinátára történő mozgásából. A teljes körű vezérlés és automatizálás érdekében elengedhetetlen volt ezen műveletek programozott végrehajtása.

Hardver szempontjából a három végrehajtó eszköz közül kettő került aktív felhasználásra, mivel a markoló egység ebben az alkalmazási környezetben nem bizonyult hasznosnak. Az írószertartó egység, egy érintésérzékeny tollal kombinálva, alkalmas érintőképernyős eszközök vezérlésére, mint például egy okostelefon. Ezzel szemben a tapadókorongos egység hatékonyan használható olyan eszközök irányítására, amelyek nyomógombokkal vagy forgatható gombokkal rendelkeznek.

A rendszer kialakítása lehetővé teszi, hogy számos különböző típusú, méretű és beviteli interfésszel rendelkező eszköz automatikus tesztelését végezzük el. Az alkalmazásba történő integráláshoz azonban néhány feltételnek teljesülnie kell: (1) az eszköz vezérlésre vagy megfigyelésre szánt felületeinek a robotkar munkaterületén belül kell elhelyezkedniük; (2) kizárólag az eszköz felső oldalán helyezkedhetnek el ezek a komponensek; (3) a bemeneti



7. ábra. A robotkar szabadságfokai a gyártó specifikációja szerint [15]

interfészeknek a robotkar számára fizikailag elérhetőnek és működtethetőnek kell lenniük. Amennyiben ezek a feltételek adottak, az eszköz, típusától függetlenül, beilleszthető a rendszerbe. A támogatott komponensek közé tartoznak az érintőképernyők, LCD vagy egyéb kijelzők, nyomógombok, tekerhető gombok, illetve más fizikai vezérlőelemek is.

A munkaterület mérete elsősorban kisebb eszközök tesztelését teszi lehetővé. Amennyiben a teszteszköz téglalap alakú, annak maximális mérete olyan kell legyen, hogy illeszkedjen a speciális szőnyegen található A , C , D , F pontok közé, ezek részletesebb ismertetésére később kerül sor. Ez körülbelül $20\text{ cm} \times 10\text{ cm}$ területet jelent, azonban az eszköz magassága tovább csökkentheti a rendelkezésre álló mozgásteret.

Egy másik lényeges korlátozó tényező a teszteszköz beviteli vezérlőinek kialakítása. A gomboknak és egyéb kezelőszerveknek olyan formával és elrendezéssel kell rendelkezniük, amely lehetővé teszi a robotkar számára az interakciót.

A robotkarhoz tartozik egy speciális szőnyeg, amely meghatározza a kar bázisának pozícióját, valamint kijelöli annak mozgásterét. Bár ezek a jelölések nem feltétlenül szükségesek a robotkar működéséhez, a szőnyegen található hat referenciapont ($A-F$) kiemelt szerepet játszik a rendszer működésében. Ezek egyrészt vizuálisan is kijelölik az alkalmazás által meghatározott mozgásteret, de legfontosabb funkciójuk a kalibráció támogatása.

A kamera képének és a robotkar saját koordináta-rendszerének összehangolása érdekében legalább négy olyan pontra van szükség, amelyek ismert koordinátákkal rendelkeznek mind a kameraképen, mind a robotkar munkaterében. A szőnyegen elhelyezett referenciapontok pontosan ezt a célt szolgálják: a robotkar, emberi közreműködés segítségével, képes kiolvasni ezek térbeli koordinátáit, miközben a kamera számára jól felismerhető objektumokként jelennek meg a felvételen. Ezáltal biztosítható, hogy a vizuális érzékelés és a robotkar mozgása pontosan szinkronizált legyen (lásd 3.4 Kalibráció alfejezet).

3.2. Kamera

A kamera elengedhetetlen része az alkalmazásnak, mivel ennek segítségével lehetséges a teszteszköz komponenseit felismerni, de a robotkar kalibrációjának során is fontos szerepet tölt be. Egy teszt lefutása során, valahányszor detektálni kell a komponenseket, a vezérlő egység lekérdezi a kamera felvétel aktuális képkockáját, amin lefuttatja az objektum-felismerő modellt. A kalibráció esetében, a korábban említett speciális szőnyegen található referenciapontokat, szintén a kamera által nyújtott képen futtatott, előre betanított, YOLOv8 modell azonosítja.



8. ábra. A tesztelőfelület amelyet a kamera felülről, egy mozdulatlan pozícióból rögzít.

Hamar bebizonyosodott, hogy a robotkarra szerelt kamera, ami a robotkarhoz járt, nem felel meg a projekt céljainak, több okból kifolyólag. Ezért az alapértelmezett kamerát hamar felváltotta egy új. Ez egy *Logitech HD Pro Webcam C910* típusú kamera, amely tágabb látószöggel rendelkezik, *FullHD*, azaz *1080p* minőségben tud rögzíteni és egy állvány segítségével egy fix pontra lehet beállítani, ami a teszt végrehajtása során nem mozdul el (lásd 8. ábra). Az eszköz érzékeny lehet kedvezőtlen fényviszonyokra, de hatványozottan jobb mint a korábbi kamera.

Az alkalmazásban több funkcionalitás is fel kívánja használni a kamera képét. Kezdetben igény szerint egy új szál jött létre, amely a kamera képrögzítését végezte. Ez a megoldás azonban problémát okozott, amikor új funkciók, például a tesztfutás rögzítése vagy az élő videó megjelenítése is igényelte a kamera képét. Mivel az operációs rendszer egyszerre csak egy csatlakozást engedélyez az eszközhöz, ezek a folyamatok blokkolták egymást.

Ezt az implementációt egy új, egyszerűbb architektúra váltotta fel. A futás során immár csak egyetlen szál kezeli a kamera képének rögzítését, és lehetőséget biztosít más folyamatok számára, hogy egy osztállyal, amely rendelkezik *on_frame()* nevű metódussal, feliratkozzanak a képkockák fogadására. Ez a függvény automatikusan meghívódik minden egyes új képkocka érkezésekor. Ezzel a megoldással a kamera kezeléséért felelős szál nem blokkolódik, miközben minimális késleltetéssel biztosítja az aktuális képet a különböző funkcionalitások számára.

Ahogy a 3. kódrészletben is látható, a *CameraFeedProvider* osztály egy Python szálból származik, és életciklusa során folyamatosan feldolgozza a kamerától érkező képkockákat, majd továbbítja azokat minden olyan osztálynak, amely fel kívánja használni. Természetesen rendelkezésre állnak olyan függvények, amelyek lehetővé teszik a képfolyamra való fel- és

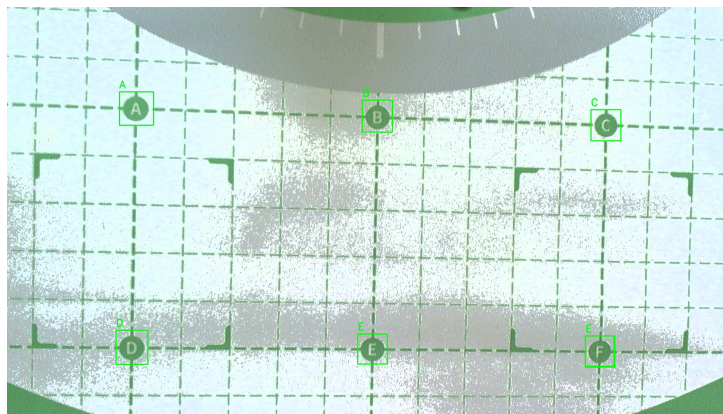
```

class CameraFeedProvider(Thread):
    ...
    def run(self):
        ...
        self.capture_started.set()
        self._running = True

        while self._running:
            ret, frame = self.camera.read()
            self.frame = frame
            ...
            for callback in self.stream_callbacks:
                callback.on_frame(frame)
            ...
        self.camera.release()
    ...

```

3. kódrészlet. A kamerát kezelő CameraFeedProvider Python osztály



9. ábra. A betanított YOLOv8 referenciapontokat felismerő modell kimenete vizualizálva

leiratkozást, a feldolgozás leállítását, a kamera váltását, illetve egyetlen képkocka lekérését is.

A kamera modul az alkalmazáson belül az említett referenciapontok detektálását is elvégzi. A betanított YOLOv8 modell egy adott képkockán történő futtatásával meghatározhatók a referenciapontok koordinátái (lásd 9. ábra).

A pontok felismerését elősegíti a bemeneti kép előfeldolgozása, ami olyan dolgokkal jár mint például a kontraszt megnövelése. Mind a hat referenciapont egyedi címkével rendelkezik, így ezek egymástól megkülönböztethetők a modell által (lásd 9. ábra). Amennyiben legalább négy különböző referenciapontot sikerül felismerni, egy validációs lépés következik. Ez azért kulcsfontosságú, mert a referenciapontok hasonló megjelenésűek, így előfordulhat, hogy a modell hibás címkét rendel hozzájuk. A validáció során ellenőrzésre kerülnek a pontok közötti távolságok és azok arányai. Például az A és C pont közötti távolságnak kétszer akkorának kell

lennie, mint az A és B pont közötti távolságnak. Ezek a megszorítások akkor is érvényesek maradnak, ha a látószög enyhén megváltozik vagy a képfelbontás módosul. Ha a validáció sikeres, a kamera modul elérhetővé teszi a detektált koordinátákat a robotkar modul számára további feldolgozás céljából.

A hibalehetőségek csökkenthetők, ha a teljes képkocka helyett csak a teszteszköz felülete kerül kivágásra és további feldolgozásra. Ezt a műveletet szintén a kamera modul végzi. A rendszer lehetőséget biztosít arra, hogy a teljes aktuális képkocka lekérése helyett kizárólag a kivágott területet használja, amelyhez két különböző implementáció áll rendelkezésre. Amennyiben a teszteszköz felülete nem található meg, a rendszer automatikusan a teljes képet adja vissza.

Az egyik megvalósítás a képen található legnagyobb körvonal azonosításán alapul. A körvonalak detektálását az *OpenCV*⁸ könyvtár *findContours()* függvénye végzi. A kapott kontúrok közül a legnagyobb kiválasztásával nagy valószínűséggel meghatározható a teszteszköz körvonala, mivel a munkafelületen ez az egyetlen objektum, amely a legkiterjedtebb körvonallal rendelkezik.

A fent említett funkcionalitás másik implementációja valójában a legnagyobb kontúrt megkereső algoritmus továbbfejlesztett változata, amelyet akkor lehet alkalmazni, ha a teszteszköz formája kevésbé emlékeztet egy téglalapra. Ez a megoldás az előző módszer kimenetét továbbfinomítja az *OpenCV* könyvtár *grabCut()* függvényének segítségével, amely a nagy kontrasztú élek mentén próbálja felosztani a képet. Az így feldolgozott kép mérete megegyezik az eredetiével, azonban a háttérként azonosított, lényegtelen pixelek elsötétítésre kerülnek. Ez biztosítja, hogy a később futtatott objektum-felismerő algoritmusok működését ne zavarják nem kívánt képelemek.

3.3. Szoftveres kapcsolat

A *Dobot Robotics* kiadott egy könyvtárat, a *DobotEDU*⁹ modult, amely lehetővé teszi a robotkar irányítását Pythonból. Azonban ez a könyvtár nem bizonyult megfelelőnek, mivel egyrészt egy régebbi Python verzióval volt kompatibilis. Másrészt a támogatott utasítások erősen korlátozottak voltak, így nem lehetett teljes mértékben kihasználni a robotkar mozgási képességeit. Mivel ez más fejlesztők számára is problémát jelentett, létrejött egy közösség

⁸Forrás: <https://opencv.org/>, utolsó megtekintés dátuma: 2025-04-22

⁹Forrás: <https://pypi.org/project/DobotEDU/>, utolsó megtekintés dátuma: 2025-04-22

által karbantartott alternatíva, a *pydobot*¹⁰ könyvtár, amely lényegesen rugalmasabb és jobban használható függvénykészletet biztosít. A *pydobot* nyílt forráskódú és MIT-licenc alatt érhető el, így gond nélkül felhasználható volt a projektben. Azonban az implementáció egy ponton nem működött megfelelően: a beépített kompresszor szívófunkciójának szabályozása során a vákuum elindítása sikeresen megtörtént, de a leállítása nem működött, így a szívás folyamatosan bekapcsolva maradt. A megoldás nem volt triviális, azonban a *Dobot Robotics* által közzétett USB-kommunikációs protokoll¹¹ segítségével sikerült azonosítani a problémát és kijavítani azt. Annak érdekében, hogy a függőségkezelő rendszer továbbra is megfelelően kezelje a projekt függőségeit, a *pydobot* *GitHub*-tárolójának egy elágaztatott verziójába feltöltésre került a szükséges javítás. Így a projekt klónozásakor automatikusan a módosított könyvtár kerül betöltésre.

A robotkar programozásánál az elsődleges szempont az volt, hogy a tesztek futtatása során a lehető legegyszerűbben lehessen használni. Ennek érdekében egy absztrakciós réteg fedti le a fentebb említett könyvtár metódusait. Az alapvető réteg biztosítja a csatlakozás létrehozását, annak állapotának ellenőrzését, valamint a robotkar mozgatásáért felelős függvények elérését. Emellett tartalmaz egy validációs mechanizmust is, amely megakadályozza, hogy a kar olyan utasításokat kapjon, amelyek a hatókörén kívülre helyeznék azt.

Egy magasabb szinten helyezkednek el a projekt-specifikus függvények. Ebben a rétegben a kalibrációs adatok globálisan tárolódnak, így a robotkar mozgatásához elegendő egy x és y koordináta megadása. Ezen a szinten történik meg a kamera képe és a robotkar koordináta-rendszere közötti szinkronizáció is, amely elengedhetetlen a pontos működéshez. Az alapvető utasítások tovább lettek fejlesztve olyan parancsokká, amelyek közvetlenül végrehajthatók a teszteszközön, például érintés, gombnyomás vagy görgetés formájában. A gombok lenyomásának kezelésére két új paraméter került bevezetésre. Az első a *deltaZ* paraméter, amely meghatározza, hogy a művelet milyen magasságból kezdődjön, ami különösen hasznos, ha a gomb kiemelkedik az eszköz felületéből. A második a *push_factor* paraméter, amely szabályozza a lenyomás mértékét, így lehetővé téve a gombok érintésének vagy teljes lenyomásának pontos beállítását. Az érintési művelet támogatja a nyomva tartást is, meghatározott időtartamig. A műveletek végrehajthatók közvetlenül pixelkoordináták alapján, a kamera képére hivatkozva, vagy valós koordináták segítségével. Amennyiben a pixelkoordináták

¹⁰Forrás: <https://pypi.org/project/pydobot/>, utolsó megtekintés dátuma: 2025-04-22

¹¹Forrás: <https://www.alcom.no/wp-content/uploads/2019/11/Dobot-Communication-Protocol-V1.1.5-1.pdf>, utolsó megtekintés dátuma: 2025-04-22

alapján történik a meghívás, az absztrakciós réteg először a megfelelő függvényekkel átalakítja azokat a valós koordináta-rendszerbe, majd a műveletet a valós koordinátákon hajtja végre.

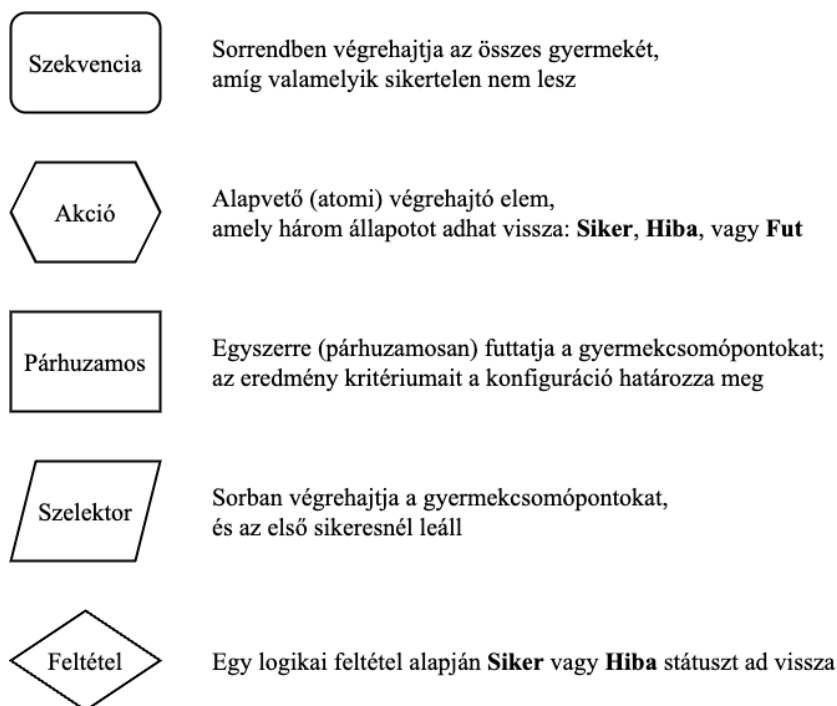
Viselkedési fa és a *py_trees*

A viselkedési fa egy nagyszerű eszköz komplex viselkedések megtervezésére.[2] Az alapötlet az, hogy a fa gyökeréből indulva minden leszármazott egy újabb viselkedési fa, és ennek segítségével összetett műveleteket lehet hierarchikus sorrendbe rendezni. A projekt során a *py_trees*¹² implementációja a viselkedési fának segített lebontani a kalibrálás és a tesztelés futtatásának folyamatait.

A viselkedési fára való igény elsősorban a rendszer komplexitásából fakadt. A kalibrációs folyamat számos egymásra épülő lépésből áll, amelyek között van logikai döntés, hibakezelés, visszacsatolás és manuális beavatkozás is. Ezek szekvenciális, illetve feltételes kapcsolatban állnak egymással, amit egy lineáris vezérlési logika nehezen kezelhetővé és átláthatatlanná tett volna.

A *py_trees* keretrendszer lehetővé teszi a folyamatok újrahasznosítható komponensekbe szervezését, valamint egyszerűsíti a hibakezelést és az állapotok követését is.

A *py_trees* keretrendszer számos előre definiált viselkedési csomópontot kínál (lásd 10.



10. ábra. A *py_trees* viselkedési fa komponenseinek típusai[13]

¹²Forrás: <https://py-trees.readthedocs.io/en/devel/>, utolsó megtekintés dátuma: 2025-04-22

ábra), valamint támogatja a naplózást és a vizualizációt, ami segített a fejlesztés és hibakeresés során. Az egyes komponensek önállóan is tesztelhetők, és könnyen integrálhatók újra a teljes fába. Ez lehetővé tette a kalibrációs logika robusztus és jól strukturált implementálását. [13]

3.4. Kalibráció

A tesztek futtatásához elengedhetetlen a robotkar pontos kalibrálása. Korán felmerült a probléma, hogy az objektum-felismerő algoritmusok által detektált komponensek pixelkoordinátái és a robotkar saját koordináta-rendszere nem kompatibilisek egymással.

Ezt a problémát projektív transzformáció alkalmazásával sikerült megoldani. Ez a módszer lehetővé teszi két sík közötti megfeleltetés létrehozását, amihez referenciapontokra van szükség, amelyek mindkét síkban ismert koordinátákkal rendelkeznek. Minél több pont áll rendelkezésre, annál pontosabb az illesztés, de legalább négy pont szükséges a transzformáció elvégzéséhez.[7] A robotkar speciális szőnyegén található hat referenciapont koordinátái a kamera modulon keresztül lekérdezhetők a kameraképből, a korábban lejegyzett YOLOv8 modell segítségével.

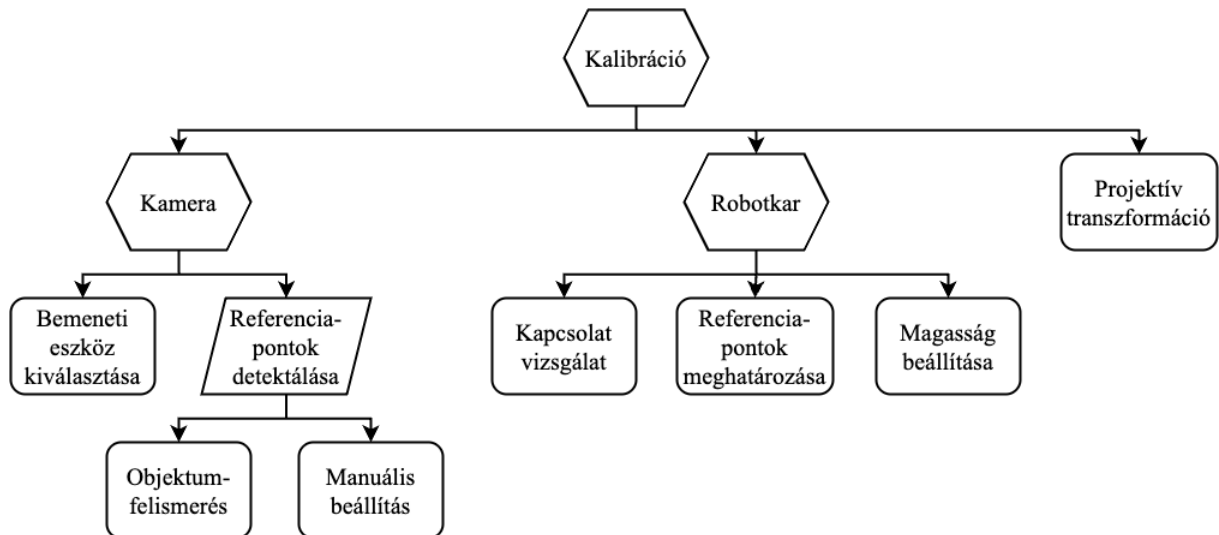
A robotkar saját koordináta-rendszerében viszont ezeket manuálisan kell regisztrálni, ami emberi beavatkozást igényel. Ez a kalibráció alapját képezi, de további manuális beállításokra is szükség van, például a tesztesszköz magasságának meghatározására. A kalibráció részeként ki kell választani a megfelelő kamerát is, ami különösen fontos, ha a szerverhez több eszköz van csatlakoztatva. Emellett a tesztelni kívánt eszközt is előre meg kell határozni a tesztek futtatása előtt. Ezek a felhasználói bemenetek alkotják a kalibrációs folyamatot.

Annak érdekében, hogy könnyen fejleszthető és moduláris legyen, a kalibráció egy viselkedési fával van vázolva, és ennek megfelelően van végrehajtva. A felsorolt okok mellett, ennek használata arra is lehetőséget ad, hogy a felhasználó számára is lépésekre lehessen bontani a kalibrációt, illetve pontos hibajelzést lehet közölni ez által.

A viselkedési fa implementálásához használt könyvtár a *py_trees*, amely segítségével könnyen monitorizálható a kalibráció különböző lépései és ezeknek sikeressége.

A 11. ábrán a kalibrációs folyamat alkotóelemei láthatók. Fentről lefelé haladva, a *Kalibráció* komponens egy szekvenciális elem, tehát csak akkor lesz sikeres, ha minden egyenes leszármazottja is sikeresen lefut balról jobbra. Így az egész folyamat a *Kamera*, *Robotkar* és *Projektív transzformáció* komponensek egymás utáni végrehajtására bontható.

A *Kamera* komponens státusza szintén az egyenes leszármazottjaitól függ, amelyek egymást követően kell helyesen végrehajtódjanak.



11. ábra. A kalibrációs folyamat döntési fája

A *Bemeneti eszköz kiválasztása* egy atomi elem, amely a felhasználó válaszára vár. Miután a kamera kiválasztásra került, elindítja a kamera inicializálását. Hiba esetén ez az információ felpropagálódik a legfelső szintre, és a kalibráció érvénytelen lesz. Ez a reakció minden esetben hasonlóan zajlik, kivéve azokat a komponenseket, amelyek képesek a hibák kezelésére.

A *Referenciapontok detektálása* egy szelektor típusú komponens, amely szintén balról jobbra hajtja végre a leszármazottait, de az első sikeres végrehajtásnál megáll és az eredményt továbbítja a felső szintre. Az ábra szerint a detektálás először objektum-felismeréssel történik, és ha ez sikertelen, manuális beállítás szükséges. Amennyiben az objektum-felismerés sikeres, nincs szükség további beavatkozásra.

Az *Objektum-felismerés* atomi elem végrehajtása során a már említett referenciapontokat detektáló YOLOv8 modell fut le. A hibák csökkentése érdekében a modell legfeljebb ötször újrafuttatódik sikertelen detektálás esetén. Ha ezt követően sem érhető el megfelelő eredmény, a komponens hibát jelez. A viselkedési fát megvalósító könyvtár lehetőséget biztosít az információ megosztására egy *blackboard* nevű kulcs-érték párokat tároló rendszeren keresztül. Ezen keresztül a viselkedési fa komponensei elérhetik a regisztrált referenciapontok koordinátáit.

A *Manuális beállítás* atomi elem vészhelyzeti megoldásként szolgál annak biztosítására, hogy a robotkar minden körülmények között kalibrálható legyen. A felhasználónak a kamerát úgy kell beállítania, hogy a kameraképre vetített referenciapontok egybeessenek a valós pontokkal. Másképpen megfogalmazva, a felhasználónak meg kell találnia egy előre meghatározott kameraszöveget és távolságot, hogy a koordináták érvényesek legyenek.

A *Kamera* komponens sikeres lefutása után következik a *Robotkar* komponens végrehajtása,

amely szintén egy szekvenciális komponens. Ennek első lépése a *Kapcsolat vizsgálat* atomi elem futtatása, amely sikeresnek tekinthető, ha a program megfelelően kommunikál a robotkarral. Amennyiben a robotkar nincs csatlakoztatva, vagy egy másik program használja, a hiba jelzésre kerül.

A *Referenciapontok meghatározása* egy manuális beavatkozást igénylő atomi elem, amely során a felhasználó segítségével rögzítésre kerülnek a referenciapontok valós koordinátái. Ez egy egyszerű folyamat, amely során a robotkar végrehajtó egységét A -tól F -ig mind a hat pontra rá kell helyezni. Ebben az esetben nem számít, hogy ez milyen magasságban történik a szőnyeghez viszonyítva, mert csak az X és Y koordináták lényegesek. Miután a robotkar egy referenciapontra ráhelyezkedett, a rendszer lekérdezi a végrehajtó egység aktuális koordinátáit, és ezeket rögzíti. A folyamat végén a beolvasott adatok szintén a közös *blackboard* tárolóba kerülnek.

A *Magasság beállítása* atomi elem hasonló az előzőhöz, azzal a különbséggel, hogy itt kizárólag a végrehajtó egység magassága a lényeges. A felhasználónak meg kell határoznia ezt az értéket a végrehajtó eszköz beállításával, majd a rendszer rögzíti azt. Ez az alapértelmezett magassági szint, amelyet a robotkar érinteni fog a műveletek során, hacsak nincs felülírva. A beállított magassági érték a robotkar számára továbbításra kerül. A *Referenciapontok meghatározása* és a *Magasság beállítása* komponensek esetében hibaellenőrzés is történik, amely kiszűri a munkafelületen kívüli értékeket.

A *Kamera* és *Robotkar* komponensek sikeres végrehajtása után minden feltétel adott a *Projektív transzformáció* elem futtatásához. Ebben a lépésben a fentebb ismertetett módszerrel létrejön a két sík közötti megfeleltetés, és felépül a fordítófüggvény, amely képes átalakítani a koordinátákat a két rendszer között. Bár a megfeleltetéshez minimum négy referenciapontra van szükség, ez még nem garantálja, hogy a transzformáció létezik. A háttérben futó egyenletrendszernek létezhet olyan kombinációja, amelynek nincs megoldása, például ha a referenciapontok egy egyenes mentén helyezkednek el. [7] Ilyen esetben a kalibrációt újra el kell végezni.

A felhasználói felületen a kalibrációs folyamat tartalmaz egy további lépést, amely során ki kell választani a teszteszközt. Ez azonban csak egy visszajelzést küld a szervernek, és nem része a viselkedési fának, mivel ez minden esetben sikeres műveletnek tekinthető.

A kalibrációs folyamat gyorsítása érdekében lehetőség van az adatok importálására és exportálására az adatbázisból. A háttérben ez ugyanúgy a viselkedési fát futtatja le a hibamentes végrehajtás érdekében, azonban a manuális interakciót szimulált módon hajtja végre.

4. Tesztek

A tesztelés fontos szerepet játszik bármilyen szoftver- vagy hardverrendszer fejlesztésében, mivel ez biztosítja, hogy az alkalmazások és eszközök az elvárt módon működjenek. A tesztelésnek az a célja, hogy azonosítsa az esetleges hibákat, biztosítva ezzel az alkalmazás hibamentes működését. A szoftvertesztelés jól bevált módszerekre és automatizált eszközökre/keretrendszerekre támaszkodhat, míg a hardvertesztelés egy sokkal összetettebb folyamatra alapszik.

A szoftverek esetében a tesztek általában virtuális környezetben futnak, ahol minden paraméter pontosan szabályozható. Ezzel szemben a hardveres tesztelés során olyan fizikai tényezőkkel is számolni kell, mint az eszközök mechanikai elemei, amelyek jelentős kihívásokat jelenthetnek.

Mindezen tesztek felépítésére egy olyan struktúrát kellett kitalálni amely képes támogatni a tesztelést érintőképernyős- vagy más hardver eszköz esetén is.

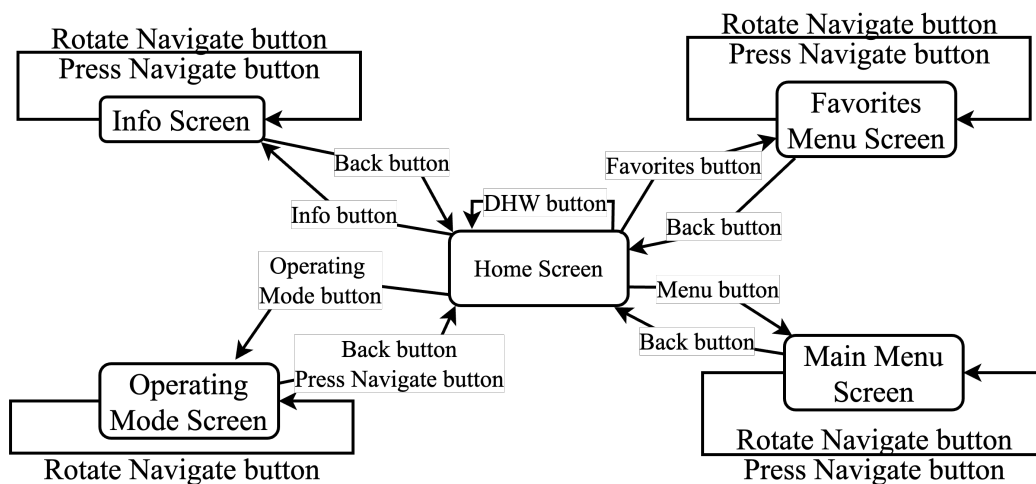
4.1. Alkalmazások és hardver eszközök működésének gráfja

Minden felhasználói interfész, legyen az egy mobilalkalmazás, egy érintőképernyős vezérlő vagy egy fizikai eszköz kezelőfelülete, egy jól meghatározott szerkezettel rendelkezik, ami meghatározza, hogyan navigálhat a felhasználó az alkalmazáson vagy rendszeren belül.

Egyes interakciós elemek (például gombok, görgők, érintőképernyős felületek) lehetőséget nyújtanak a felhasználónak, hogy a rendszer egy adott állapotából egy másikba jusson el. Ezek az állapotok tekinthetők különböző oldalaknak vagy felhasználói felületeknek, amelyek között a felhasználó egy adott művelet révén mozoghat. Például a *Vissza* gomb megnyomásával az aktuális oldalról egy korábbi állapotba térhet vissza, míg egy *Tetszik (Like)* gomb nem változtat a navigációs állapoton, de módosítást végez az adott oldalon (például egy elem állapotának frissítésével).

Tanulmányozások után az az ötlet született, hogy a navigációt lehetővé tevő struktúra hatékonyan ábrázolható gráf formájában, függetlenül attól, hogy egy adott művelet valóban állapotváltást eredményez-e, vagy csupán egy módosítást hajt végre az aktuális állapoton belül.

Ez a megközelítés lehetővé teszi az alkalmazások és hardver eszközök működésének strukturált leképezését, valamint segíti az automatizált tesztek definiálását és végrehajtását. Az állapotok (oldalak) a gráf csomópontjaiként értelmezhetők, míg az interakciós elemek,



12. ábra. Fűtőrendszerek kezelésére használt controller navigálási gráfja.

amelyek különböző műveleteket hajtanak végre, például navigációt, gombnyomást vagy beállítások módosítását, a gráf éleit alkotják. Ezzel a módszerrel vizuálisan és logikailag is jól követhető módon modellezhető, hogy a felhasználó milyen lehetőségekkel rendelkezik egy adott rendszerben, és hogyan léphet interakcióba az egyes elemekkel (lásd fűtőrendszerek vezérlésére használt controller navigálási gráfja, 12. ábra).

A gráfalapú ábrázolás nemcsak az alkalmazások és hardver eszközök működésének strukturált leképezését teszi lehetővé, hanem hozzájárul az automatizált tesztek hatékony megtervezéséhez és végrehajtásához is. Mivel ilyen típusú ábrázolás segítségével pontosan meghatározhatók a tesztelési útvonalak, könnyebben azonosíthatók a potenciális hibák, és biztosítható, hogy minden interakció lefedésre kerüljön a tesztfolyamat során.

Az implementációt tekintve, minden alkalmazás esetén ábrázolni kell a navigációs struktúrát, ezért olyan megoldást kellett találni, amely könnyen módosítható, rugalmas, és nem jelent korlátozásokat a bővítések, új funkciók hozzáadása vagy az alkalmazások újrastrukturálása során.

A nyári gyakorlat alatti fejlesztés során a projekt nem rendelkezett adatbázissal, és az adatbázis integrálása nem tartozott a projekt prioritásai közé. Így szükség volt egy olyan megoldásra, amely az adatokat a memóriában tárolja. A megoldás a strukturált adatfájlok alkalmazása volt, mint a *JSON*, *XML* és *YAML* formátumok.

A három közül a *JSON* formátum tűnt a legalkalmasabbnak és legfelhasználóbarátabbnak a fejlesztés idején. *JSON* formátumban könnyedén definiálhatóak a csomópontok (oldalak) és élek (interakciós elemek) amelyek segítségével az alkalmazás vagy rendszer navigációs fája

pontosan és érthetően ábrázolható. Ezzel biztosítható, hogy a rendszer könnyen módosítható és bővíthető legyen, miközben az adatok jól strukturáltak és egyszerűen kezelhetők maradnak. Ennek bővítését jelenleg csak fejlesztők tudják elvégezni, ugyanis a felhasználóknak nincs lehetőségük módosítani ezen állományokat a felhasználói interfészen keresztül.

4.2. Tesztek felépítése

A tesztek, amelyeket a felhasználó létrehozhat, lépésekből épülnek fel. Minden lépés a gráfban található éleket reprezentálja, azaz olyan műveleteket, amelyek tesztelik az adott teszteszköz funkcionalitását. Például egy gomb helyes működésének ellenőrzése vagy egy interakciós elem válaszának vizsgálata tartozhat ide. Az egyes lépések összessége alkotja a teljes tesztet. Mivel a tesztek lépéseit a gráf struktúrája határozza meg, azok automatikusan illeszkednek a rendszer navigációs folyamatához és lehetőséget adnak a funkcionalitás tesztelésére.

Egy lépést három fő paraméterrel lehet leírni a felhasználó számára. Az első az interakciós művelet típusa, amely meghatározza, hogy a robotkar milyen típusú interakciót fog végrehajtani az eszközzel. Az interakciós műveletek közé tartozik az érintés/lenyomás (kontroller gombjai esetén), görgetés vagy a forgatás.

A második paraméter az interakciós elem neve, amely lehetővé teszi, hogy az objektum-felismerő modellek azonosítani tudják a tesztelni kívánt komponenst. Ez lehet egy gomb vagy bármilyen más interaktív komponens, amelyre az adott művelet irányul.

A harmadik paraméter, a művelet iránya, opcionális jellegű. Nem minden interakciós művelet igényel irányt, de egyesek esetén, mint például a görgetés vagy a forgatás, szükséges meghatározni, hogy a művelet mely irányba történjen. Ez az irány lehet függőleges (fel-le) vagy spirális (jobbra-balra).

A navigálási gráf JSON objektumaiban tárolt paraméterek nemcsak azokat az adatokat tartalmazzák, amelyeket a felhasználónak közvetlenül szükséges ismernie, hanem egyéb, háttér-információkat is, amelyek a rendszer működéséhez elengedhetetlenek, de nem szükségesek a felhasználó számára. Ezek az információk minden eszköz esetén egy külön *app_structure.json* fájlban vannak tárolva, amelyekben két fő objektum-lista van: *nodes* (csomópontok) és *edges* (élek).

A *nodes* objektum-listán belül más objektumok vannak tárolva amelyeknek van egy *id* és egy *name* mezőjük, ahol az *id* egy egyedi azonosítót jelent, és a *name*, az oldal neve egy

Mező	Leírás
id	Az él egyedi azonosítója, amely lehetővé teszi az egyértelmű azonosítást.
componentName	Az interaktív komponens neve, amely az alkalmazás interfészén található.
from	A csomópont (oldal), ahonnan a navigálás elindul.
to	A csomópont (oldal), ahová a navigálás történik.
static	Logikai érték: statikus (true) él esetén nincs navigáció, míg dinamikus (false) esetén van.
deltaHeight	Az érintés függvényének paramétere, amely az alapértelmezett végrehajtási magasságot módosítja (lásd 3.4. alfejezet).
push	Logikai érték: igaz esetén lenyomás szükséges, hamis esetén elég az érintés.
action	Az interakció műveletének neve.
direction	A művelet iránya (pl. up, down a görgetéshez).
hold	Az érintés időtartama másodpercben.

3. táblázat. Az él objektum mezőinek leírása.

alkalmazás/interfészen belül.

Az *edges* objektum-listán belül vannak az élek tárolva (lásd él objektum mezői 3. táblázat).

A tesztek, amelyeket a felhasználók létrehozhatnak, úgy vannak tervezve és felépítve, hogy ne csak szakmabeliek vagy programozási ismeretekkel rendelkező felhasználók legyenek képesek ezeket létrehozni. A cél az, hogy a tesztelési folyamat mindenki számára elérhető és könnyen kezelhető legyen. Ennek érdekében a felhasználói felület úgy van megtervezve, hogy intuitív módon, minimális technikai tudás birtokában bárki képes legyen tesztek létrehozni, módosítani és végrehajtani (lásd 6. fejezetben).

Minden eszköz esetén a felhasználó feladata, hogy a robotkar végrehajtó egységét felszerelje, ugyanis, nem minden teszteszköz esetében felel meg ugyanaz a végrehajtó egység, és nem minden végrehajtó egység képes minden műveletet végrehajtani, de az biztosítva van, hogy egy eszköz teszteléséhez csak egy végrehajtó egység típus legyen használva, így nem kell tesztelés közben cserélni az egységeket.

A *FestivApp* mobilalkalmazás tesztelésekor az érintésérzékeny toll van használva, amely lehetővé teszi az érintés alapú interakciókat érintőképernyőkön, így pontosan szimulálhatóak a felhasználói műveletek, például a képernyő érintése és görgetése. A fűtőrendszerek kezelésére használt kontroller esetén a tapadókorongos egység van használva. Ez az egység nemcsak az érintési műveleteket képes elvégezni, hanem forgatásra is alkalmas, amely lehetővé teszi a

különböző irányokban történő műveleteket, például forgatás jobbra vagy balra.

A tesztek helyes végrehajtásának biztosítása érdekében egy helyességi feltételt (assertion) kellett bevezetni. Ez az ellenőrzési mechanizmus a navigálási gráf alapján dönti el, hogy a teszt futtatása utáni oldalon milyen interakciós elemeknek kell jelen lenniük.

4.3. Tesztek futtatása

A tesztek és a hozzájuk tartozó helyességi feltételeket a backend szerver *JSON* formátumban kapja meg és dolgozza fel.

Az első lépésben a rendszer validálja a kapott *JSON* adatstruktúrát, amely tartalmazza a teszteszköz nevét, a tesztlépéseket és a helyességi feltételeket is. A rendszer ellenőrzi, hogy minden szükséges mező helyesen szerepel-e. Hiányzó vagy hibás adatok esetén a teszt nem indítható el.

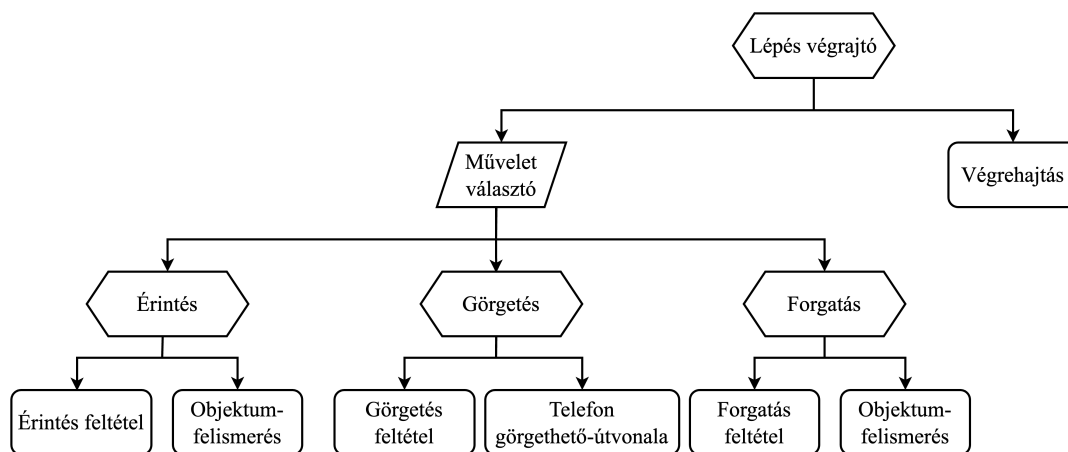
A következő fázisban a rendszer ellenőrzi a tesztlépések helyes sorrendjét a navigációs gráf alapján. Mivel minden lépés egy adott oldalra vezet, majd onnan újabb navigációs lehetőségek állnak rendelkezésre, az ellenőrzés során biztosítani kell, hogy a lépések logikai sorrendje megfeleljen az alkalmazás navigációs struktúrájának. Ez megakadályozza, hogy egy teszt érvénytelen útvonalakat próbáljon bejárni.

Az ellenőrzések után következik a teszt végrehajtása, ahol a teszthez tartozó lépések kerülnek külön végrehajtásra. Minden tesztlépéshez egy viselkedési fa (behaviour tree) van rendelve, amely felelős a kamera által biztosított képen az objektumok felismeréséért és a kívánt művelet végrehajtásáért. A viselkedési fa egy hierarchikus struktúra, amely lehetővé teszi a komplex döntési folyamatok lebonyolítását, és minden lépést egy-egy döntési pontként kezel (lásd 13. ábra).

A döntési fa alkalmazása azért előnyös, mert ha egy lépés végrehajtása során bármelyik része nem sikerül (például egy objektum nem kerül felismerésre, vagy egy interakció nem hajtható végre), akkor az egész lépés végrehajtása sikertelennek minősíthető. Ezáltal a teszt nem folytatható tovább, mivel a lépés nem hajtódott végre helyesen. A rendszer egyértelműen jelzi, ha valamelyik lépés nem a kívánt módon hajtódott végre.

A *Lépés végrajtó* egy szekvenciális komponens, amely az összes direkt alárendelt elemét, a *Művelet választó*-t és az *Végrehajtás*-t futtatja. Ha ezen komponensek közül bármelyik nem hajtódik végre helyesen, akkor az egész lépést sikertelennek tekinti.

A *Művelet választó* egy szelektor komponens, amely az első sikeres direkt alárendelt



13. ábra. A lépések végrehajtásának döntési fája.

elem futtatása után megáll. Ez lehetővé teszi, hogy a rendszer eldöntse, melyik művelethez tartozik az adott lépés. A *Művelet választó* komponens három alárendelt elemmel rendelkezik: *Érintés*, *Görgetés* és *Forgatás*. Mindhárom szekvenciális komponens, amelyek közül a rendszer a támogatott műveletek alapján választja ki, hogy melyik művelet hajtsódjon végre, és a szükséges paramétereket is beállítja.

Ezeknek a műveleteknek a végrehajtása során az *Érintés*-, *Görgetés*- és *Forgatás feltétel* komponensek visszatérhetnek siker vagy hiba állapotokkal, amelyek eldöntik, hogy melyik művelet végrehajtása folytatódjon.

A *Érintés* és *Forgatás* műveletek esetén egy interfész komponenst kell felismerni a teszteszközön. Ehhez a feladathoz az *Objektum-felismerés* csomópont felel, amely az ún. *blackboard* kulcsok segítségével beállítja a komponens megfelelő koordinátáit. Ezeket a koordinátákat későbbi csomópontok olvashatják ki, és felhasználhatják az adott művelek végrehajtásához.

A *Görgetés* esetében nem szükséges komponenst felismerni, hanem az eszköz pozícióját, az ún. *Telefon görgethető-útvonala*-t kell meghatározni. Ehhez két koordináta szükséges, amelyek azt az útvonalat írják le, ahol a görgetés műveletet végre kell hajtani. Számítógépes látás alkalmazásával a telefon képernyőjének keretét körbe lehet rajzolni, és a távolságvizsgálat segítségével meghatározható az a szakasz, ahol a felhasználó általában görget. Ezen feladat megoldásához az *OpenCV* *fitLine* függvénye kerül alkalmazásra. Ebben az esetben is a szakasz meghatározásához szükséges két koordináta a *blackboard* kulcsok segítségével kerül eltárolásra.

Miután a *Művelet választó* sikeresen végrehajtotta a művelethez szükséges feladatokat, a

Végrehajtás csomópont kiolvassa a *blackboard*-ról, a görgetés koordinátáit (ha léteznek), és a komponens középpontjának a koordinátáit, amelyeket az *Objektum-felismerés* csomópontok állítottak be. Minden művelet esetén a robot modulhoz tartozó megfelelő függvények vannak meghívva.

4.4. Tesztek ellenőrzése

A tesztek helyességének ellenőrzéséhez szükség van előre meghatározott feltételekre, amelyek biztosítják, hogy a teszt sikeressége értékelhető legyen. Ezek a feltételek lehetővé teszik, hogy a felhasználó anélkül döntsön egy teszt eredményéről, hogy visszanézné a tesztelési folyamatot videón vagy személyesen figyelné a futtatást.

A rendszer lehetőséget biztosít testre szabott helyességi feltételek létrehozására is. A felhasználó kiválaszthatja azokat az interfészelemeket, amelyeknek kötelezően jelen kell lenniük a teszt végrehajtása után, illetve azokat, amelyeknek nem szabad megjelenniük. Amennyiben a felhasználó nem definiál egyedi feltételeket, a sikeresség alapértelmezett kritériuma az, hogy az elvárt interakciós elemek legalább 60%-a felismerhető legyen a végrehajtás után.

Ez a megközelítés hatékony mobilalkalmazások tesztelésénél, azonban az új eszköz – fűtőrendszerek vezérlésére használt kontroller – bevezetésével a módszer már nem bizonyul kellően megbízhatónak. Ennek oka, hogy az ilyen eszközök esetében az interakciós elemek fizikai gombok, amelyek mindig jelen vannak, függetlenül a képernyőn megjelenő tartalomtól.

Ennek a problémának a megoldására *OCR* (optikai karakterfelismerő) algoritmusok vannak alkalmazva, amelyek képesek képek alapján szöveges információt kinyerni. A felhasználó minden lépéshez megadhat egy elvárt szöveget, amelynek a lépés végrehajtása után meg kell jelennie a tesztelendő eszköz képernyőjén. Így lehetőség nyílik arra, hogy a rendszer automatikusan ellenőrizze, valóban a kívánt állapotba került-e az eszköz.

A legjobban teljesítő megoldás az *EasyOCR*, egy nyílt forráskódú *Python* könyvtár, amely több nyelven is támogatja a karakterfelismerést. Mivel az *OCR* algoritmusok nem mindig 100%-os pontosságúak – például eltérő betűtípusok, képernyőminőség vagy tükröződés esetén – kiegészítésként a *FuzzyWuzzy* könyvtár is használva van. Ez lehetővé teszi két szöveg összehasonlítását, és százalékos értékkel jelzi, mennyire egyeznek. A gyakorlatban 80%-os hasonlóság felett van a szöveget elfogadottnak tekintve. Ez az érték számos tesztelés alapján lett meghatározva, figyelembe véve a felismerési hibákat és a valós alkalmazási környezetet.

A szövegfelismerés pontosságának növelése érdekében a képeket több lépéses

előfeldolgozáson mennek keresztül, mielőtt az *OCR* algoritmus lefutna. A bemeneti képet először szürkeárnyalatossá lesz alakítva, majd *CLAHE* (*Contrast Limited Adaptive Histogram Equalization*) módszerrel növelve van a kontraszt, hogy a szöveg jobban elkülönüljön a háttértől. Ezután élesítő szűrő van alkalmazva, amely tovább kiemeli a karaktereket, végül a kép felnagyításra kerül, hogy az *OCR* nagyobb felbontásból dolgozhasson. Ezt követően az *EasyOCR* könyvtár elvégzi a szövegfelismerést, és az így kinyert szöveg összehasonlítva lesz az elvárt értékkel. Az ilyen előfeldolgozás jelentősen javítja a felismerés megbízhatóságát, különösen alacsony minőségű vagy zajos képek esetén (szövegfelismerés implementációt lásd 4. kódrészlet).

Lehetőség van a teszteket importálni és exportálni is a felhasználó gépére. Ilyenkor egy *JSON* formátumú fájl jön létre, amely tartalmazza a teszt kezdő állapotát, a lépések sorozatát, az egyes lépésekhez tartozó alternatívákat, valamint a használt eszközt. Ez a funkció lehetővé teszi a tesztek egyszerű mentését, megosztását vagy újrafuttatását.

```
def get_text_from_image(image):
    gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
    clahe = cv2.createCLAHE(clipLimit=2.0, tileGridSize=(8, 8))
    contrast_enhanced = clahe.apply(gray)
    kernel = np.array([[0, -1, 0], [-1, 5, -1], [0, -1, 0]])
    sharpened = cv2.filter2D(contrast_enhanced, -1, kernel)
    processed_image = cv2.resize(
        sharpened, None, fx=2, fy=2, interpolation=cv2.INTER_CUBIC
    )
    reader = easyocr.Reader(["en"])
    results = reader.readtext(processed_image)
    return " ".join([text[1] for text in results])

def check_text_existence_in_image(
    image,
    text,
    confidence=HIGH_CONFIDENCE,
):
    data = get_text_from_image(image)
    if fuzz.partial_ratio(text, data) / 100.0 >= confidence:
        return True
    return False
```

4. kódrészlet. Szöveg kinyerése és hasonlósága egy megadott szöveggel.

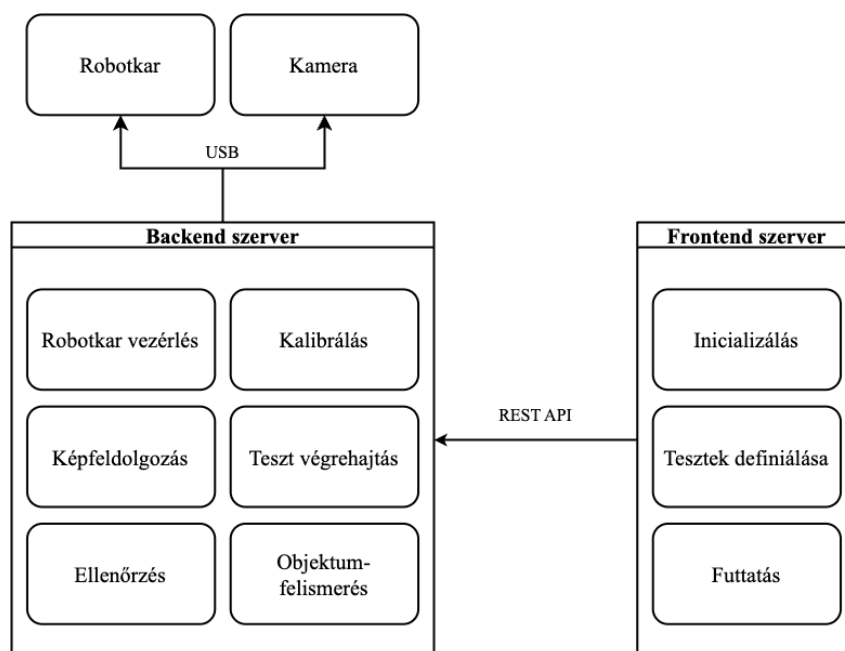
5. Architektúra

Az alkalmazás két különálló rétegből áll: egy feldolgozó rétegből és egy megjelenítési rétegből. A feldolgozó réteg tartalmazza az összes objektum-felismeréssel és vezérléssel kapcsolatos funkcionalitást, míg a megjelenítési réteg, *HTTP* kéréseken keresztül kommunikálva, jeleníti meg a felhasználó számára releváns funkciókat (lásd 14. ábra). A feldolgozó réteg futtatása során kiemelten fontos figyelembe venni a hardveres erőforrásokat, mivel ez a réteg nagyobb számítási kapacitást igényel. Ez egyrészt a modellek futtatásából, másrészt a több szálon történő végrehajtásból adódik. A rétegek egyértelmű szétválasztásának köszönhetően a megjelenítési réteg nem támaszt különösebb hardverkövetelményeket, így könnyen kitelepíthető. A rétegek közötti kommunikáció *REST API*-t használ, kivételt képez ez alól a videó felvételek továbbítása, ugyanis a gyorsaság érdekében ez *socket*-eket alkalmaz.

5.1. Feldolgozó réteg

A feldolgozó réteg (backend) Python nyelven készült, mivel a robotkar programozása ebben a nyelvben lehetséges, és a legkorszerűbb mesterséges intelligencia eszközök, például a YOLOv8, szintén ebben érhetőek el.

Ez a réteg számos egyéb funkciója mellett *API*-t is biztosít a megjelenítési réteg számára, így (backend) szerverként működik. Mivel a megjelenítési réteg egy webes felület, a webszerver



14. ábra. A rendszer architektúrája

alapú megoldás bizonyult a leghatékonyabbnak kommunikációs szempontból. A választás *Flask*-re esett, mivel ez a keretrendszer minden szükséges funkcionalitást biztosít, miközben nem terheli feleslegesen az alkalmazás futását. A robotkar és a kamera kizárólag USB-kábellel csatlakozik a szerverhez, ami jelentősen megnehezíti a rendszer távoli telepítését.

A webszerver számos funkcionalitást biztosít a megjelenítési réteg számára, amelyeket megfelelően kellett strukturálni. A *REST* konvenció bizonyult a legmegfelelőbb választásnak, mivel logikus és jól szervezhető keretet biztosít a funkciók számára. Emellett a *JSON* formátum természetes és kézenfekvő megoldást kínál az adatok továbbítására.

Az adatok perzisztens tárolására egy *PostgreSQL* adatbázis áll rendelkezésre. Erre esett a választás, mivel hatékonyan képes tárolni a *JSON* formátumú adatokat, és kiválóan kompatibilis a *Python* nyelvvel. A táblák és azok kapcsolatai egy egyed-kapcsolat diagram segítségével kerültek megtervezésre, amelyet könnyen rá lehetett vetíteni a relációs adatbázisra. Az adatbázisban többek között a következő táblák találhatók: *Tests* (tesztek), *Configs* (betölthető kalibrációk) vagy *Subjects* (teszteszközök). A szerver csatlakozhat egy lokálisan futó adatbázishoz, vagy a könnyebb orkesztráció érdekében egy *Docker*-konténerben futó példányhoz, amelyet környezeti változókkal lehet konfigurálni.

5.2. Megjelenítési réteg

A projekt során kiemelt figyelmet kapott a korszerű technológiák és megoldások integrálása. Ennek eredményeként a megjelenítési réteg webalkalmazásként valósult meg, asztali alkalmazás helyett. Ez lehetővé tette a feldolgozó és megjelenítési réteg teljes szeparálását, valamint rugalmasabb és könnyebben hozzáférhető felhasználói élményt biztosított.

A webalkalmazás *JavaScript* nyelven íródott, és a *React* könyvtár segítségével egy intuitív, letisztult felület készült. Mivel az alkalmazás nem kizárólag szakemberek számára készült, a felhasználói felület kialakítása során ez figyelembe volt véve.

A megjelenítés fejlesztése során lehetőség nyílt egy dizájnér bevonására a felület megtervezéséhez. Ennek eredményeként egy *Figma* tervezőprogramban elkészített dizájn vázlat született, amely alapján a weboldal megvalósult. A szükséges *React*-komponensek a *Bootstrap* könyvtár által biztosított elemek testreszabásával készültek el.

A webalkalmazás funkcionalitásai kizárólag bejelentkezett felhasználók számára érhetők el. A felhasználómenedzsmentet egy privát *Keycloak* szerver biztosítja, amelyet a *Codespring* bocsát a projekt rendelkezésére.

6. Alkalmazás működése

Annak érdekében, hogy a teszt megfelelően fusson le, és az objektum-felismerés pontosan működjön, bizonyos környezeti feltételeknek teljesülniük kell. Az optimális felismerési teljesítmény érdekében fontos, hogy a teszteszközt egyenletes megvilágítás érje minden irányból, minimalizálva az árnyékokat és a fényvisszaverődéseket. Közvetlen fényforrás nem lehet a teszteszköz felett, mivel a visszaverődő fény akadályozhatja az objektumok felismerését. A képernyő fényereje sem lehet túl magas vagy túl alacsony; az optimális tartomány 45–80% közé esik. Az eszközt úgy kell elhelyezni, hogy a robotkarral ellentétes irányba nézzen, lehetőleg a munkaterület közepén, biztosítva a kamera számára az egyenletes és zavartalan rálátást.

A tesztelések során megfigyelhető volt, hogy ha egy fényforrás közvetlenül a teszteszköz felett helyezkedik el, és erős visszaverődést eredményez, az jelentősen csökkentheti az objektum-felismerés hatékonyságát. Ilyen esetekben az interakciós elemek felismerése pontatlanná válhat, vagy teljesen sikertelen lehet. Emiatt fontos, hogy a fényforrások úgy legyenek elhelyezve, hogy ne okozzanak zavaró tükröződést a teszteszköz felületén. Emellett a kamera fókusz- és expozíciós beállításai is befolyásolják a felismerési teljesítményt. Ha például egy mobiltelefon képernyője túl erős fényerővel rendelkezik egy gyengén megvilágított környezetben, akkor a kamera nehezen tud megfelelően fókuszálni, ami elmosódott vagy túlexponált képet eredményezhet. Ilyen esetekben az objektumok még emberi szemmel is nehezen azonosíthatók a képen, ami megnehezíti az automatikus elemzést is.

A megfelelő működés érdekében minden támogatott teszteszközhöz egy dedikált végrehajtó eszköz szükséges, így a tesztelés közben nincs szükség végrehajtó eszközcserére. Mobil alkalmazások, például a *FestivApp* teszteléséhez egy írószert rögzítő végrehajtó eszköz használatos, amely egy érintésérzékeny tollat tartalmaz, lehetővé téve a pontos és megbízható interakciót a kijelzőn. A fűtőrendszerek vezérlésére szolgáló kontroller esetében egy tapadókorongos végrehajtó egység biztosítja a stabil kapcsolatot a fizikai gombokkal.

Kalibrációs folyamat

A webalkalmazáson keresztül lehetőség van a robot kalibrációjának elvégzésére, amely elengedhetetlen lépés a tesztelési folyamat megkezdéséhez, hiszen kalibráció nélkül a robotkar nem képes pontos mozgásokat végrehajtani, és a tesztelés sem futtatható megfelelően. A kalibráció egy négy lépésből álló folyamat, amely biztosítja, hogy a rendszer pontosan érzékelje

a tesztelendő környezetet, és megfelelően végezze el az interakciókat.

Az első lépés a kamera kiválasztása, amely során meg kell határozni azt a kamerát, amely optimálisan rálát a tesztfelületre. Ez különösen fontos, mert a kamera képe alapján történik az objektumok felismerése és a referencia pontok beazonosítása.

A kamera kiválasztásával egy időben a tesztfelület szőnyegén található referencia pontok felismerése is megtörténik. Ha az automatikus pontfelismerés sikertelen vagy pontatlan eredményt ad, akkor manuális pontmegfeleltetés szükséges. Ez azt jelenti, hogy a felhasználónak a kamerakép alapján manuálisan kell igazítani a szőnyegen található pontokat a rendszer által kijelölt referencia pontokhoz. Ennek célja, hogy a kalibráció a lehető legpontosabb legyen, és a robotkar mozgása megfelelően a valós környezet geometriájának.

A második lépés a pontok kalibrációja, amely során a fizikai szőnyegen található pontok koordinátája kerül rögzítésre. Ebben a lépésben a felhasználónak a robotkart minden egyes pont fölé kell helyeznie, és igazolnia kell annak megfelelő elhelyezkedését. A pontok precíz kalibrációja kulcsfontosságú, mivel ez határozza meg a robotkar mozgási koordináta-rendszerét, így biztosítva, hogy a későbbi tesztelés során minden művelet megfelelően végrehajtható legyen. Ezt a lépést összesen hat referencia pontra kell végrehajtani, így garantálva a pontos és stabil kalibrációt (lásd 15. ábra).

A harmadik lépés az eszköz magasságának beállítása, amely meghatározza a tesztelendő eszköz domináns magassági szintjét. Mivel különböző eszközök eltérő magassággal rendelkeznek, ez a lépés elengedhetetlen a robotkar megfelelő pozicionálásához. A robotkart a tesztelendő eszköz főbb szerkezeti elemeinek magasságába kell helyezni, majd a rendszerbe el kell menteni az értéket. Ennek a beállításnak a célja, hogy a robotkar megfelelően alkalmazkodjon az eszköz geometriájához, és minden interakciós műveletet a megfelelő szinten hajtson végre.

Az utolsó lépés a tesztelendő eszköz kiválasztása, ami lehetőséget ad a megfelelő tesztet létrehozni. A kiválasztott eszköztől függően különböző tesztelési módszerek alkalmazhatók. Ebben a lépésben a rendszer felkínálja a rendelkezésre álló eszközök listáját, amelyek közül a felhasználó kiválaszthatja a megfelelőt. Az eszköz kiválasztása után lehetőség nyílik a megfelelő tesztek konfigurálására és végrehajtására, biztosítva, hogy a tesztelési folyamat a kívánt eszközhöz és környezethez igazodjon.

A teljes kalibrációs folyamat elvégzése után a robot készen áll a tesztelési folyamatok végrehajtására. Minden kalibráció után lehetőség van ennek elmentésére, így nem szükséges

2

Point Calibration

You have to move the robot arm to the specified point and click "OK" button when you are ready. Try to place it as accurate as possible.

A	✓
B	✓
C	✓
D	↺

OK

15. ábra. Pont kalibráció lépés.

minden esetben új kalibrációt végrehajtani, hanem ezt importálni is lehet.

Teszt felvezetése

A teszt futtatása egy kezdőképernyő kiválasztásával kezdődik, amely során meg kell adni, hogy a tesztet milyen állapotból kell elindítani. Ez egy kulcsfontosságú lépés, hiszen a teszt pontos végrehajtása érdekében az eszközt a kívánt állapotba kell hozni.

A kezdőállapot kiválasztása után lehetőség van a teszt lépéseinek hozzáadására. Ezeket a lépéseket a rendszer a robotkaron keresztül fogja végrehajtani, biztosítva az automatikus tesztelést. A lépések hozzáadásakor megadható, hogy milyen műveleteket kell végrehajtani és milyen objektumra. Minden lépéshez opcionálisan megadható egy elvárt szöveg is, amelynek a végrehajtás után meg kell jelennie a tesztelendő eszköz képernyőjén. Ez a szöveg ellenőrzési feltételként szolgál, és automatikusan kiértékelésre kerül, hogy a lépés helyesen hajtott-e végre (lásd 16. ábra).

Test Execute for AirWaterController

Using the dropdown menu, please select the starting page of the test, and after that, select steps to build a test.

Selected page Home

Step 7

ROTATE Nav RIGHT

Temperature

Step 8

ROTATE Nav RIGHT

+ Add Text Assertion

TAP Back

Add Step

Run



16. ábra. Teszt lépések létrehozása.

A teszt végrehajtásának megkezdése előtt lehetőség van helyességi feltételek (lásd 4.4 Tesztek ellenőrzése alfejezet) megadására is. Ezek a feltételek biztosítják, hogy a teszt megfelelően ellenőrizze a rendszer működését. Ezeket a feltételeket a felhasználó manuálisan is beállíthatja, vagy a rendszer automatikusan végez egy helyességi ellenőrzést a háttérben. A teszt végrehajtása során a felhasználó élő kameraképen követheti a folyamatot, amely megmutatja, hogyan hajtja végre a robotkar az egyes lépéseket.

A teszt befejezése után egy eredményösszesítőt kapunk, amely tartalmazza a végrehajtott lépések számát, valamint azt, hogy a helyességi feltételek teljesültek-e. Az eredmények vizuálisan is megjelennek, és a felhasználó egy videófelvételt is letölthet vagy újra megtekinthet a teszt végrehajtásáról. Ez a funkció különösen hasznos lehet a hibakeresés és a tesztfolyamat optimalizálása során.

7. Következtetések és továbbfejlesztési lehetőségek

A dolgozatban bemutatott rendszer lehetőséget biztosít a felhasználó számára, hogy egy interakciót igénylő eszközt teszteljen robotkar segítségével, objektum-felismerésre támaszkodva. A tesztelési folyamatot egy webes alkalmazáson keresztül lehet elérni, amely biztosítja az alkalmazás nyújtotta funkciók elérését. A robotkar kalibrálása után a felhasználó képes egy többlépéses tesztet végrehajtani. A rendszer minden teszthez és lépéshez egyedileg megadható ellenőrzési feltételeket kínál, amelyek automatikusan értékelik a teszt sikerességét.

A rendszer alapjai jól használhatónak bizonyultak, azonban a megbízhatóság növeléséhez további fejlesztésekre van szükség. Emellett több olyan funkcionális is beépíthető, amelyek a felhasználói élményt növelik, valamint közelebb viszik a piacképes állapothoz. A következő továbbfejlesztési lehetőségek kiemelendők:

- Tesztkonfigurációk importálása és exportálása adatbázisból, valamint egy áttekinthető felület (dashboard) kialakítása, ahol a felhasználó megtekintheti a korábban futtatott tesztek és azok eredményeit.
- Statisztikák készítése a lefuttatott tesztek és azok eredményei alapján, például sikerességi arány, gyakori hibák, átlagos futási idők stb.
- Lehetőség biztosítása több teszt egymás utáni, automatizált lefuttatására.
- Egy olyan rendszer kialakítása, amely lehetővé teszi a felhasználó számára új teszteszközök felvezetését. Ez magába foglalná a szükséges konfigurációs fájlok generálását, valamint új modellek betanítását is, lehetőleg minimális emberi beavatkozással.
- A backend szerver telepíthetőségének megvalósítása, valamint a robotkarral való kommunikáció leválasztása a kábeles kapcsolatról.

Hivatkozások

- [1] A. Ahmed és mások. „Toward a New Methodology for an Efficient Test of Reconfigurable Hardware Systems”. *IEEE Transactions on Automation Science and Engineering* 15 (2018), 1864–1882. oldal. DOI: 10.1109/TASE.2018.2822050.
- [2] O. Biggar, Mohammad Zamani és I. Shames. „An Expressiveness Hierarchy of Behavior Trees and Related Architectures”. *IEEE Robotics and Automation Letters* 6 (2021), 5397–5404. oldal. DOI: 10.1109/LRA.2021.3074337.
- [3] James Gallagher Brad Dwyer. *Getting Started with Roboflow*. 2023. URL: <https://blog.roboflow.com/getting-started-with-roboflow/> (elérés dátuma: 2025. ápr. 3.)
- [4] Arindam Chaudhuri és mások. „Optical Character Recognition Systems”. *Optical Character Recognition Systems for Different Languages with Soft Computing*. Cham: Springer International Publishing, 2017, 9–41. oldal. ISBN: 978-3-319-50252-6. DOI: 10.1007/978-3-319-50252-6_2. URL: https://doi.org/10.1007/978-3-319-50252-6_2.
- [5] Brandon J. Dilworth, A. Karlicek és L. Thibault. „An Approach to Component Testing: An Analytical Study”. *Sensors and Instrumentation, Aircraft/Aerospace, Energy Harvesting & Dynamic Environments Testing, Volume 7* (2019). DOI: 10.1007/978-3-030-12676-6_31.
- [6] Tanmay Diwan, G. Anirudh és J.V. Tembhurne. „Object detection using YOLO: challenges, architectural successors, datasets and applications”. *Multimedia Tools and Applications* 82 (2023. márc.), 9243–9275. oldal. DOI: 10.1007/s11042-022-13644-y. URL: <https://doi.org/10.1007/s11042-022-13644-y>.
- [7] Elan Dubrofsky. „Homography estimation”. *Diplomová práce. Vancouver: Univerzita Britské Kolumbie* 5 (2009).
- [8] Sunila Gollapudi. „OpenCV with Python”. *Learn Computer Vision Using OpenCV: With Deep Learning CNNs and RNNs*. Berkeley, CA: Apress, 2019, 31–50. oldal. ISBN: 978-1-4842-4261-2. DOI: 10.1007/978-1-4842-4261-2_2. URL: https://doi.org/10.1007/978-1-4842-4261-2_2.
- [9] Ultralytics Inc. *Ultralytics YOLO Docs*. 2025. URL: <https://docs.ultralytics.com/> (elérés dátuma: 2025. ápr. 3.)
- [10] Licheng Jiao és mások. „A Survey of Deep Learning-Based Object Detection”. *IEEE Access* 7 (2019), 128837–128868. oldal. DOI: 10.1109/ACCESS.2019.2939201.
- [11] P Kumar és E Manash. „Deep learning: a branch of machine learning”. *Journal of Physics: Conference Series* 1228 (2019. máj.), 12045. oldal. DOI: 10.1088/1742-6596/1228/1/012045.

- [12] D. Piumatti és mások. „An efficient strategy for the development of software test libraries for an automotive microcontroller family”. *Microelectronics Reliability* 115 (2020), 113962. oldal. DOI: 10.1016/j.microrel.2020.113962.
- [13] *Py Trees*. URL: <https://py-trees.readthedocs.io/en/devel/index.html> (elérés dátuma: 2025. ápr. 7.)
- [14] Sarbjeet Singh Ravpreet Kaur. „A comprehensive review of object detection with deep learning”. (2023).
- [15] Shenzhen Yuejiang Technology Co., Ltd. *Dobot Magician Lite User Guide (DobotLab-based)*. Shenzhen Yuejiang Technology Co., Ltd. Floor 9-10, Building 2, Chongwen Garden, Nanshan iPark, Liuxian Blvd, Nanshan District, Shenzhen, Guangdong Province, China, 2022. URL: [https://download.dobot.cc/product-manual/magician-lite/en/Dobot%20Magician%20Lite%20User%20Guide%20\(DobotLab-based\).pdf](https://download.dobot.cc/product-manual/magician-lite/en/Dobot%20Magician%20Lite%20User%20Guide%20(DobotLab-based).pdf).
- [16] Jiameng Shi és mások. „Facilitating Non-Intrusive In-Vivo Firmware Testing with Stateless Instrumentation”. *Proceedings 2024 Network and Distributed System Security Symposium* (2024). DOI: 10.14722/ndss.2024.23116.
- [17] U. Sirisha és mások. „Statistical Analysis of Design Aspects of Various YOLO-Based Deep Learning Models for Object Detection”. *International Journal of Computational Intelligence Systems* 16.1 (2023), 126. oldal. ISSN: 1875-6883. DOI: 10.1007/s44196-023-00302-w. URL: <https://doi.org/10.1007/s44196-023-00302-w>.
- [18] Jacob Solawetz. *What is Mean Average Precision (mAP) in Object Detection?* 2020. URL: <https://blog.roboflow.com/mean-average-precision/> (elérés dátuma: 2025. ápr. 19.)
- [19] Satoru Takekoshi, Takahiro Shinagawa és Kazuhiko Kato. „Testing device drivers against hardware failures in real environments”. *Proceedings of the 31st Annual ACM Symposium on Applied Computing* (2016). DOI: 10.1145/2851613.2851740.
- [20] Ultralytics. *Datasets for YOLOv5, YOLOv8 and other Ultralytics models*. <https://docs.ultralytics.com/datasets/>. (Elérés dátuma: 2025. ápr. 19.)